M T U - 1 3 0

MACASM

6502 MACRO ASSEMBLER

Release 1.1

U S E R M A N U A L

October, 1982

COPYRIGHT NOTICE

Micro Technology Unlimited
2806 Hillsborough Street
P.O. Box 12106
Raleigh, NC 27605 USA
(919) 833-1458

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

No warranties, either express or implied, are made by Micro Technology Unlimited with respect to this manual or the software described herein, its quality, performance, merchantability, or fitness for any particular application. This product is sold "as is". The buyer assumes all risk as to quality and performance. Under no circumstances will MTU be liable for direct, indirect, incidental or consequential damages resulting from any defect in the software, even if MTU has been advised of the possibility of such damages. Should the software prove defective following purchase, the buyer assumes the entire cost of all necessary servicing, repair or correction and any incidental, indirect, or consequential damages. Additional rights vary from state so some of the above exclusions and limitations may not apply to you.

NOTICE

Micro Technology Unlimited reserves the right to make changes to the product and specifications described in this manual at any time without notice.

TABLE OF CONTENTS

APPENDICIES

MTU 6502 MACASM MACRO ASSEMBLER

INTRODUCTION

The MTU macro assembler (MACASM) is a disk-based standard two-pass assembler which runs on the MTU-130. It accepts 6502 assembly language source programs as input and produces 6502 machine-language programs as output, along with a formatted listing. The source program to be assembled should be a standard ASCII text file on disk, and can be easily generated using the MTU Editor. When the assembler is run, the machine language object code is written to a disk file as a standard CODOS command file, which can subsequently be executed by merely typing the name of the file. The listing of the program is normally output to a file, but can be easily redirected to a printer or disabled completely.

The MTU assembler accepts standard MOS-Technology (Commodore) format source programs. This is the industry standard format for 6502 assembly language and is the format used in virtually all textbooks on 6502 assembly language. Extensions for complete support for macros and conditional assembly are provided. Macros may have up to 36 arguments of any length and may be nested and invoked recursively. A file of macro definitions which allows programs for 8080 or Z-80 microprocessors to be assembled using MACASM is supplied on the distribution disk (see appendix F).

It is quite practical to develop very large and complex programs using MACASM, since programs with several thousand lines can be assembled in only a couple of minutes. Files can be as large as necessary, up to the full capacity of a disk. Multiple source files can be read automatically during assembly. The combination of full-screen editor, assembler, and CODOS breakpoint capability makes the MTU-130 the finest development system for 6502 assembly language software available on the market today. In fact, the software engineers at MTU assembled all the MTU-130 software, including the CODOS operating system (about 10,000 lines) using MACASM on the MTU-130, leaving the cross assembler on their 16-bit computer unused and forgotten!

The Listing produced by the assembler includes a complete alphabetically-sorted symbol table and a cross-reference map. The cross reference map identifies every occurrence of a symbol in the program, in sorted order. Once you have used this map, you will wonder how you ever got along without it! An error summary is displayed on the console of the computer during assembly, so that you will normally not need to consult the listing to correct errors. Naturally, errors are also flagged in the listing. If desired, the error summary can be suppressed or diverted to a printer or file.

Two special files called the "Definitions" file and the "Externals" file can be used to link separately-assembled programs together. Another unique capability of MACASM is "direct definitions" which can be used to tell the assembler to "relocate" a program without modifying the source program. Because MACASM outputs object code to disk instead of directly to memory, there are no "reserved" areas of memory where programs cannot be assembled. A single object code file can contain any number of discontinuous memory blocks, all of which can be loaded with a single CODOS command. An assembler directive is provided to specify in which memory bank a block of code is to assembled, and different banks may be used in the same file. In software development applications where it is necessary to download the object code to another computer, the SERDMP Utility program can be used to output papertape-format object code.

This manual assumes that the you already have a working knowledge of 6502 assembly language programming skills. If you don't meet this prerequisite, you will want to consult one of the many tutorial works available, some of which are listed in the References.

SOURCE PROGRAM REQUIREMENTS

The input to the assembler is one or more files of text on disk, normally prepared using the MTU Editor. Each line of the file may hold one assembly language statement, an assembler directive, or a comment.

Comments

Any line which contains no characters, or only blanks is interpreted as a comment and is ignored by the assembler except that it becomes part of the listing. Any line whose first non-blank character is a semi-colen (;) is also a comment. Comments may be included at the end of an assembly language statement or assembler directive. Anything between a semicolen and the end of line is considered a comment (exception: semicolens may be embedded in a character string, described later). In addition, if characters follow one or more blanks terminating the last operand of a statement, they will be treated as a comment even if no semicolon is present.

Identifiers

An Identifier is a name devised by the programmer to symbolically identify an address, value, or macro. An identifier consists of from one to 31 characters. The first character must be an upper case alphabetic character. The remaining characters must be chosen from the following set:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _ . ?

Note that only upper case alphabetic characters are acceptable. If you wish to keep your symbols compatible with other 6502 assemblers for the sake of "portability" between systems, you should be aware that most other systems only allow identifiers of up to six characters and will not accept "_", "." or "?" characters. On some other systems, longer identifiers are permitted but only the first six or eight characters are "significant". For example, the identifiers INVENTORYREAD and INVENTORYWRITE would be considered to be the same identifier by some assemblers. On the MTU assembler, all characters are "significant", so ASM will correctly distinguish between INVENTORYREAD and INVENTORYWRITE. Many programmers like to use the characters "_"(underline), "." and "?" to improve the readability of composite names. MACASM has one "reserved" identifier which is the single character "A". This identifier is predefined to mean the 6502 accumulator register and must not be used as an ordinary identifier. Machine language opcode mnemonics (e.g., LDA, JSR, etc.) may be used as ordinary identifiers. Most other systems prohibit this so you may wish to avoid it for the sake of portability. Macro names may duplicate another identifier used as an ordinary label, but may not duplicate a standard opcode mnemonic. The following are legal identifiers:

B	DATA	AA	WRITE.12	A_5...9?CODE
JSR	ZZZZ	THIS_IS_A_LONG_IDENTIFIER		WHAT????

The following are ILLEGAL identifers (for the reasons indicated in parentheses):

Exit	(can't use lower case letters)
45RPM	(must start with an alphabetic character)
HEX$D	($ is not a legal character)
A	(A is a reserved identifier)
THIS-IS	("-" is not a legal character)
JSR	(illegal if used as macro name; legal if used as an ordinary label)

FIELDS

An assembly language statement is divided into as many as four different fields, each of which may or may not be present. The first field, if present, is the Label Field and must start in column 1. If column 1 is a blank then the label field is not present. If column one is non-blank (and not a comment), then the Label field is present and must contain an identifier. If a label is present, no other fields are required, but usually will be present.

The second field is the Opcode field, which is optional but normally present. The Opcode field begins in the first non-blank column following the Label field if present. If no label is present, it begins in the first non-blank character, starting with column 2. The Opcode field must contain a legal 6502 assembly language mnemonic, assembler directive, or the name of a previously defined macro. Standard opcodes and assembler directives are listed in Table 1 and Table 2. Macros will be discussed later. If you have previous experience with the Rockwell AIM-65 assembler, you should note that the opcode cannot start in column 1 on the MTU assembler as it can for the AIM.

The third field is the Operand field. It may or may not be present, depending on the Opcode field. The Operand field is separated from the Opcode field by one or more blanks. The requirements for the contents of the Operand field depends on the Opcode and is discussed later.

The fourth field is the comment field, which may or may not be present at the option of the programmer. If present, it is separated from the Operand field either by one or more blanks or by a semicolon or both. In the case of an Opcode which does not permit an operand field, the comment field can be similarly separated from the Opcode field. The assembler recognizes lines of up to 80 characters. Lines longer than 80 characters should not be used.

To improve the readability of assembly language programs, most programmers adopt a convention of placing each of these fields in a fixed column. Many programmers use the following:

Columns 1-6	Label field
Columns 8-13	Opcode field
Columns 16-30	Operand field
Columns 32-62	Comments

We suggest that you limit your statements to 62 characters. If you do, then the listing generated by the assembler will be limited to 80 characters wide so that you can print it on the MTU-130 console or an 80-column printer without having lines "wrap around". The listing has 18 columns of additional information added to each source line by the assembler, making a total of 80.

Some sample statements with comments indicating the kinds of fields present are given below:

```
BEGIN                              ;Label field only with comment
        TXA                        ;Opcode field only with comment
;       Comment only
        JSR     ONE                ;Operand and Opcode fields with comment
MY.XY   LDA     (THERE+2),Y        ;Label, Opcode, and Operand fields with comment
```

TABLE 1: STANDARD 6502 OPCODE MNEMONICS

ADC	Add with carry
AND	AND with A
ASL	Arithmetic shift left
BCC	Bcranch if carry clear
BCS	Branch if carry set
BEQ	Branch if equal
BNE	Branch if not equal
BMI	Branch if minus
BPL	Branch if plus
BVC	Branch if overflow clear
BVS	Branch if overflow set
BIT	Test bits with A
BRK	Break
CLC	Clear carry
CLD	Clear decimal mode
CLI	Clear Interrupt disable
CLV	Clear overflow flag
CMP	Compare with A
CPX	Compare with X
CPY	Compare with Y
DEC	Decrement memory by one
DEX	Decrement X by one
DEY	Decrement Y by 1
EOR	Exclusive OR with A
INC	Increment memory by one
INX	Increment X by 1
INY	Increment Y by 1
JMP	Jump
JSR	Jump to subroutine
LDA	Load A
LDX	Load X
LDY	Load Y
LSR	Logical shift right
NOP	No operation
ORA	OR with A
PHA	Push A
PHP	Push flags
PLA	Pull A
PLP	Pull flags
ROL	Rotate left
ROR	Rotate right
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract with borrow
SEC	Set carry
SED	Set decimal mode
SEI	Set interrupt disable
STA	Store A
STX	Store X
STY	Store Y
TAX	Transfer A to X
TAY	Transfer A to Y
TSX	Transfer S to X
TXA	Transfer X to A
TXS	Transfer X to S
TYA	Transfer Y to A

ADDITIONAL 6502 OPCODE MNEMONICS SUPPORTED BY ASM

BLT	Branch if less than (equivalent to BCC)
BGE	Branch if greater than or equal (equivalent to BCS)
SVC	Supervisor call (equivalent to BRK followed by .BYTE with a single numeric operand evaluating to an 8 bit quantity)

TABLE 2: ASSEMBLER DIRECTIVES

=	Equate
==	Equate override
*=	Set program counter
.BANK	Specify memory bank
.BYTE	Define byte data
.DATE	Embed date of assembly in code
.DBYTE	Define double byte data
.DEF	Declare definition
.ELSE	Else for conditional assembly
.END	End of assembly
.ENDMAC	End of macro definition
.ENDIF	End block of cond. assembly
.ENTRY	Declare entry point
.FILL	Define block of constant data
.IF	Conditionally assemble if true
.IFEQ	Conditionally assemble if equal
.IFGE	Cond. asm. if greater or equal
.IFGT	Cond. asm. if greater than
.IFLE	Cond. asm. if less than or equal
.IFLT	Cond. asm. if less than
.IFNE	Cond. asm. if not equal
.IFNULL	Cond. asm. if no operand present
.LIST	Turn listing on or off
.MACRO	Define a macro
.OPT	Declare options
.OVL	Specify CODOS overlay number
.PAGE	Eject to top of page
.READ	Read alternate file
.WORD	Define address data

OPERANDS AND ADDRESSING MODES

MACASM supports standard 6502 assembly-language syntax for operands and addressing modes, with some extensions. The format for the 13 possible addressing modes of the 6502 processor are given below and in the Wirth syntax diagrams in Appendix C. Appendix B lists all legal instruction/addressing mode combinations.

Operand form	Addressing mode(s) specified
	Implicit (no operand)
A	Accumulator
#n	Immediate
nn	Absolute or relative
nn,X	Absolute, X
nn,Y	Absolute, Y
n	Zero page
n,X	Zero page, X
n,Y	Zero page, Y
(xx)	Indirect
(n,X)	Indirect, X
(n),Y	Indirect, Y

where:

n is an expression which evaluates to an 8-bit quantity
nn is an expression which evaluates to a 16-bit quantity
xx is an expression which evaluates to an 8 or 16 bit quantity

Unlike some 6502 assemblers which require you to explicitly indicate zero page addressing mode using non-standard syntax, the MTU assembler will automatically select zero page addressing mode when appropriate. No blanks may be embedded in the operand. ASM will flag illegal addressing modes as an error.

EXPRESSIONS

The MTU assembler allows unusually sophisticated expressions to be used in operands, which will be evaluated by the assembler. Appendix C illustrates a syntax diagram for forming a legal expression.

Numeric values may be specified in decimal, hexadecimal, octal or binary form. The default is decimal. A prefix must be placed on numeric data to specify a different radix, as follows:

$ indicates a hexadecimal number
@ indicates an octal number
% indicates a binary number

Some examples of legal numeric values are:

```
0      $D2     @177   $0200  %00010000  32147  $FFFF
```

Some examples of ILLEGAL numeric values are:

```
1,655  ;(comma is not permitted)
$-100  ;(sign is not permitted after $)
2D00   ;(D is an illegal character in a decimal number)
$ 0A   ;(blank is not permitted between the prefix and the number)
```

MACASM has an unusually powerful set of arithmetic operators available to the programmer for use in expressions. These operators are shown in Appendix C and below:

+	addition
-	subtraction
*	multiplication
/	division
\	remainder
&	bit-by-bit AND
¦	bit-by-bit OR
~	bit-by-bit exclusive OR
<	low byte only
>	high byte only

A very unusual feature of the MTU assembler is that numeric values and expressions are evaluated by the assembler using 24 bit signed arithmetic. This gives you great freedom in forming expressions, especially when using multiply and divide operators, without worrying about overflow. For example the expression

```
480*256/8
```

would overflow producing erroneous results on most assemblers but evaluates correctly (to 15360) on the MTU assembler. The assembler can process values in the range of -8388608 to +8388607 decimal.

The assembler evaluates expressions from left to right without precedence of operators except for ⟨and⟩ which have lower precedence than all other operators. Therefore a ⟨or⟩ operator operates on the entire expression that follows it, but the unary - operator only operates on the value which immediately follows it. Square brackets [and] may be used to show grouping of operations in the same way that parentheses are normally used. Parentheses may not be used to show grouping because of possible ambiguities since parentheses indicate indirect addressing modes. Brackets may be nested up to three deep. Some example expressions are:

```
25              ;evaluates to 25 decimal
-3+4            ;evaluates to 1
17-$A           ;evaluates to 7
3+2*6           ;evaluates to 30 (not 15!)
3+[2*6]         ;evaluates to 15
>$B3A0+$0100    ;evaluates to $B4 (not $01B3!)
[>$B3A0]+$0100  ;evaluates as $01B3
$B3A0/$100+$100 ;evaluates as $01B3
10¦%01000000    ;evaluates as 74 decimal (%01001010)
```

As shown in the syntax diagram in Appendix C, values used in the expression may be numbers, identifiers, single-character strings, or the program counter symbol. When an identifier is used, the assembler substitutes the value of the identifier in the expression. Single character constants can be used in expressions, in which case the ASCII value of the character is substituted. Character constants are denoted by a single quote prefix ('). Another single quote may optionally follow the character. Examples:

```
'A'             ;evaluates as 65 decimal ($41)
' '+1           ;evaluates as 33 decimal ($21)
'1-1            ;evaluates as 48 decimal ($30)
```

The symbol * can be used in expressions to represent the present value of the program counter. The present value of the program counter means the address of the first byte of the statement currently being assembled. The fact that the program counter symbol is the same as the multiplication operator will not pose any problem for the assembler. For example:

```
*+3             ;evaluates to the program counter plus three
3**+1           ;evaluates as three times the program counter plus 1
***             ;evaluates as the program counter squared!
```

The example program segments below illustrate some common usages of operand expressions. Ellipsis (...) represent some intervening statements which are not shown.

```
NITEMS  =       20              ;MAXIMUM NUMBER OF "ITEMS"
ITMSIZ  =       4               ;NUMBER OF BYTES PER ITEM
TABSIZ  =       NITEMS*ITEMSIZ  ;TABLE SIZE TO HOLD ALL ITEMS
;
        *=      $0010           ;PROGRAM COUNTER IS $0010 IN ZERO PAGE
PRT1    *=*+2                   ;RESERVE 2 BYTES FOR POINTER
RESULT  *=*+1                   ;RESERVE 1 BYTE FOR RESULT
        ...
        *=      $1000           ;PROGRAM COUNTER TO $1000 FOR MAIN PROGRAM
START   LDA     #<TABLE
        STA     PTR1            ;INSTALL LOW BYTE OF ADDRESS OF TABLE INTO PRT1
        LDA     #>TABLE
        STA     PTR1+1          ;INSTALL HIGH BYTE OF ADDRESS
        ...
        LDA     RESULT
        LDY     #11*ITMSIZ-ITMSIZ
        STA     (PTR1),Y        ;INSTALL RESULT IN 11TH ITEM OF TABLE
        ...
        *=*+256/256*256         ;START TABLE ON NEXT EXACT PAGE BOUNDARY
TABLE   *=*+TABSIZ              ;RESERVE ROOM FOR TABLE
        ...
```

FORWARD REFERENCES

A forward reference is an operand containing an identifier which is not defined until later in the program being assembled. Forward references are permitted for all normal instructions and most assembler directives. However, forward references to zero page values should be avoided. That is, all identifiers which evaluate to less than $0100 should be defined before they are used in the program. Since instructions referencing zero page are two bytes long and instructions referencing non-zero page instructions are three bytes long, the assembler will always assume a non-zero page value and generate 3-byte code for any operand which is not yet defined when referenced. Thererfore a forward reference to a page 0 quantity may result in the generation of an absolute (3 byte) form of code. If the instruction requires a zero-page operand (e.g., indirect,X addressing), and the operand contains a forward reference, an error message will be generated. Most programmers define all zero-page values before any other part of the program.

ASSEMBLER DIRECTIVES

Assembler directives, also called pseudoinstructions or simply pseudos, are instructions which tell the assembler to perform certain actions. The directives available on the MTU-130 are listed in Table 2. The directive mnemonic should be used in the Opcode field in the same manner as for actual machine instructions. Some directives require operands and some do not. Each of the directives available is discussed in detail below.

The Equate Directive

The equate (=) directive can be used to define the numeric value of an identifier. It is frequently used to define constants used throughout the program. An equate MUST have a label which is the identifier being defined, and requires an operand which must evaluate to a numeric value. Some examples of legal equates are:

```
START   =       0
M6522=$BFC0
PORTAD=   M6522+1
VMSIZE  =480*256/8
BLANK   =       ' '
COD_ORG = $C000+VMSIZE
```

The operand may contain forward references (that is, it may refer to another identifier which is subsequently defined in the program). This permits constructions such as:

```
PROGSIZE=END-START
        ...
START   CLD
        ...
        LDX  #PROGSIZE
        ...
END     RTS
```

The equate override directive (==) is discussed later in the section describing the ASM command line options.

The Program Counter Directive

The program counter directive (*=) tells the assembler to begin assembling instructions at the address specified in the required operand field. Normally a program counter directive will appear before any statements generating code in a program. The assembler will begin assembling code starting at address $0000 if no *= is specifed. The operand MUST be fully evaluatable when it is encountered in pass 1 (that is, it may not contain any forward references). Besides defining the starting address for assembly, the program counter is used to reserve uninitialized storage for variables. Examples:

```
        *=   $4200
DATA1   *=   *+1        ;ONE BYTE VARIABLE
STUFF   *=*+2           ;TWO BYTE VARIABLE
ASAVE   *=*+1
```

You should be careful when reserving space for variables to remember that you may not have any embedded blanks in the operand, because the assembler will treat everything after the first blank as a comment. For example,

```
VALUE   *=      *+3
```

correctly reserves three bytes of space, but

```
VALUE   *=*     +3
```

will not reserve any space because the assembler treats the "+3" as a comment!

You may have any number of *= directives in a program. Each time the assembler encounters a *= directive followed by some code (not just reserved undefined storage), it will output another CODOS loadable record to the object file. The assembler does not do any checking for conflicting use of memory, so it is entirely the programmer's responsibility to ensure that code following the *= does not overlap other code in another block. It is not necessary to define blocks of code in ascending order. For example it is legal to have:

```
        *=$4000
        ...
        *=$2000
        ...
```

.BYTE Directive

The .BYTE directive is used to assemble one or more bytes of initialized data storage. The .BYTE directive must have at least one operand, and may optionally have a label. Multiple operands are permitted, separated by commas. Each operand can either be an expression which evaluates to a numeric value or a character string enclosed in single quotes ('). The assembler generates one byte of data for each character in the string. Some examples of legal .BYTE directives are given below with the comments indicating the code which will be generated.

```
NUMS    .BYTE   1               ;GENERATES $01
        .BYTE   0,-1            ;GENERATES $00, $FF
        .BYTE   'ABC'           ;GENERATES $41,$42,$43
ABCS    .BYTE   'ab','c'+$80    ;GENERATES $61,$62,$E3
        .BYTE   13,'BCD',0      ;GENERATES $0D,$42,$43,$44,$00
```

Lower case strings are permitted. If a numeric operand evaluates to a number which cannot be represented in a single byte, only the low order 8 bits of the value will be generated, without any error message. Example:

```
TOOBIG  .BYTE   $213            ;GENERATES $13
```

When multiple operands are specified, blanks may be inserted between the comma and the next argument to improve readability. However, no blanks must be present in front of the comma, or the assembler will treat the second operand as a comment. For example:

```
        .BYTE  10,    20,    30
```

generates three bytes, but,

```
        .BYTE   10      ,20     ,30
```

generates only one, because the assembler will treat the first blank after "10" as the delimiter of the operand field.

.WORD Directive

The .WORD directive is used to generate one or more "words" of data. Actually the term "word" is somewhat of a misnomer since the word length of the 6502 is one byte, but it is widely used in the context of assembly language to mean a two-byte quantity, such as an address, stored in memory with the low order byte first. The .WORD directive requires at least one operand and may optionally have a label. Each operand must evaluate to a numeric value. Forward references are permitted. Example:

```
        *=      $2000
        .WORD   $570B           ;GENERATES $0B AT $2000 AND $57 AT $2001
HERE    .WORD   HERE,THERE,*    ;GENERATES $02,$20,$08,$20,$02,$20
THERE   .WORD   'A',13          ;GENERATES $41,$00,$0D,$00
        ...
```

If an operand evaluates to a value which cannot be represented in a 16-bit quantity, then only the low order 16-bits of the value will be generated, without error message. Example:

```
        .WORD   $FFFF+4         ;GENERATES $03,$00
```

Multiple operands may have blanks between the comma and the next operand, as described for .BYTE above.

.DBYTE Directive

The .DBYTE directive generates two-byte data in the same manner as .WORD above, except that the data is stored with the most significant byte first and the least significant byte at the next higher address. For example:

```
        .DBYTE  $570B           ;GENERATES $57 FOLLOWED BY $0B
```

.FILL Directive

The .FILL directive is used to define a repetetive block of data. The .FILL directive must have two operands. The first operand is a repeat count specifying the number of times the second operand is to be repeated. The second argument is treated exactly the same as a .BYTE operand. Therefore,

```
        .FILL   4,0
```

is equivalent to

```
        .BYTE   0,0,0,0
```

The second operand may take any form that is legal for .BYTE operands. For example:

```
NENTRIES =  100
        ...
TABLE   .FILL   NENTRIES,'                    ',$0D,0
```

will generate 2200 bytes of code consisting of 100 repetitions of a string of 20 blanks followed by $0D and $00. If a .LIST 3 or .LIST 7 directive was given previously, the listing will display the address of each of the 100 occurrances. Normally only the first occurrance would be shown.

.DATE Directive

The .DATE directive installs the current 9 character date as entered when the MTU-130 was powered up at the current progam counter location. This is useful for embedding the creation date of a program in the object code. .DATE has no operands. For example:

```
SIGNON  SVC     2
        .BYTE   2,'STARTREK VERSION 6,789,200.1 '
        .DATE
        .BYTE   ' BY MEE TU',0
        ...
```

When SIGNON is executed, it will display somehting like:

```
STARTREK VERSION 6,789,200.1 9-SEP-82 BY MEE TU
```

.PAGE Directive

The .PAGE directive is used to tell the assembler to advance to the top of the next page of the listing and print the header. The .PAGE directive may optionally include a single string operand of up to 45 characters enclosed in single quotes. This string will be used for the title of the new page. Example:

```
        .PAGE   'OPERATING SYSTEM Interface Routines'
```

MACASM automatically starts a new page with the most recent page title after 57 lines of printout are generated on the listing (see appendix D for alterations).

.LIST Directive

The .LIST directive is used to instruct the assmbler to turn the listing on or off. It can optionally have one operand which evaluates to a numeric value. If no operand is present, the listing is turned on beginning with this statement. If an operand is present, it is evaluated. If the value is zero, the listing is turned off starting with the next statement. If it is non-zero, the listing is turned on starting with this statement. The listing may be turned on and off any number of places in the file. .LIST directives will have no effect if the listing is globally turned off by an L=N command on the ASM command line (discussed later). Example:

```
OFF     =       0
;
        .LIST   OFF             ;TURN OFF LISTING
        ...
        .LIST                   ;TURN LISTING BACK ON
```

Other numeric values may be specified as the operand in order to govern the way macro invocations are listed and the way lines with multiple operands (such as .BYTE) are listed. The value of the operand is really treated as a set of one-bit flags, where each flag has meaning as shown below.

Bit #	Meaning if bit=1
0	Listing enabled
1	Print values of multiple argument pseudos on additional lines
2	Print expanded lines of macros.

The default value is 1, which is the standard listing without printing additional lines for multiple argument pseuods, and without printing macro invocations. Here are all the sensible combinations:

```
.LIST     0          ; STOP PRINTING LISTING
.LIST     1          ; PRINT NORMAL LISTING
.LIST     3          ; PRINT MULTI-ARGUMENTS, DONT EXPAND MACROS
.LIST     5          ; EXPAND MACROS, DONT PRINT MULTI-ARGUMENTS
.LIST     7          ; EXPAND MACROS, PRINT MULTI-ARGUMENTS
```

More information on macro listing is given in a succeeding section.

.END Directive

The .END directive is used to identify the last line to be assembled in a file. No operands are used. If no .END is present in the file being assembled, the assembler will read until end-of-file is encountered. No error will be generated. When the assembler encounters a .END in a file specified on a .READ directive, it will stop reading the specified file but continue assembling beginning with the line following the .READ in the original file. This allows a file to be assembled either as a stand-alone program or as an "include" file for a larger program without modification.

.OPT Directive

The .OPT directive is ignored by the MTU assembler. It is included in order to keep compatibility of source decks with programs generated for other assemblers, which use the .OPT directive to specify assembly options such as listing disposition, object code suppression, etc. On the MTU assembler these options are specified in a much simpler and more convenient manner as arguments on the MACASM command line (discussed later). By allowing assembly options to be specified on the MACASM command instead of using an .OPT directive, you can specify your options without having to edit the source program.

.ENTRY Directive

The .ENTRY directive is used to declare the entry point to the main program being assembled. A .ENTRY directive does not have an operand. Only one .ENTRY directive can appear in an assembly. If no .ENTRY is specified, the entry point defaults to the address of the first byte of code generated by the assembler. The entry point is the address to which CODOS will transfer control when the program is executed. The .ENTRY directive should be located immediately before the desired entry point. Example:

```
        *=      $1000
BUFEND  .WORD   THERE
        ...
        .ENTRY
BEGIN   CLD
        ...
```

.BANK Directive

The .BANK directive is used to declare the memory bank number in which the following code should be generated. .BANK requires one operand which evaluates to a numeric value between 0 and 3. In the absence of .BANK directives, all code is generated so that it will be loaded in bank 0. Example:

```
        ...
        .BANK   1
        *=      $C000           ;ASSEMBLE THE FOLLOWING IN THE DISPLAY MEMORY
        .BYTE   1,2,3,4,5,6
        ...
        .BANK   0
        *=      $1000           ;MAIN PROGRAM IN BANK 0
        ...
```

.OVL Directive

The .OVL directive is used to declare that the following code should be assembled in an object code record with a specified overlay number. The operand should evaluate to a number between 0 and 255. Since the current version of CODOS does not support automatic overlay loading, this directive is included only for purposes of compatibility with possible future products. See Appendix B of the CODOS manual for further information.

.READ Directive

The .READ directive is used to declare the name of another source file which is to be inserted into the assembly. The .READ directive must have one operand which is a valid CODOS file name, optionally with a drive number. This file is sometimes called an "include" file. The default file extension is ".A". The default disk drive number is the drive that the original source file resides on.

```
        .READ   MYSUBS          ;Include file MYSUBS.A.
        .READ   STD_DEFS:1      ;Include STD.DEFS.A on drive 1.
```

When the assembler encounters a .READ directive, it suspends reading of the current source file and instead accepts input from the specified file. Reading continues from that file until the assembler encounters a .END, End-of-File, or another .READ. If a .END or End-of-File is encountered, the assembler frees the file and resumes reading the original source file beginning with the statement following the .READ. Note that encountering a .END does not terminate the assembly but merely terminates input from the specified file. .READ statements can be "nested" up to three deep, meaning that a file specified on a .READ may itself have .READ directives. Any number of .READS may be included in a file. The .READ directive is very useful for including files of frequently-needed definitions, macros, or subroutines in assemblies, and for breaking very large programs up into more manageable pieces.

.DEF Directive

The .DEF directive declares the names of identifiers whose definitions are to be added to the Definitions file at the completion of assembly. The use of the .DEF Directive is discussed in the section on the MACASM command line arguments. A .DEF Directive must have at least one operand which is a valid identifier defined elsewhere in the program. Multiple operands may be specified if separated by commas. The names may not be macros. Example:

```
        .DEF    DISKIN
        .DEF    FFTGO,FFTEND,MATINV,U0
```

The assembler will append the definition of each identifier declared in a .DEF directive to the end of the Definitions file in the form of an "equate override". For example, the assembler might add the following based on the example .DEF above:

```
DISKIN==$13B5
FFTGO==$2100
FFTEND==$21CC
MATINV==$A23A
U0==$00B0
```

CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly allows the assembler to selectively ignore portions of the input source file if a specified condition is not met. The block of code to be conditionally assembled is initiated by one of the following directives and terminated by an .ENDIF directive:

.IF	assemble if operand is non-zero
.IFLT	assemble if 1st operand is less than second operand
.IFLE	assemble if 1st operand is less than or equal to second operand
.IFNE	assemble if 1st operand is not equal to second operand
.IFEQ	assemble if 1st operand is equal to second operand
.IFGE	assemble if 1st operand is greater than or equal to second operand
.IFGT	assemble if 1st operand is greater than second operand
.IFNULL	assemble if no operand is present

In the following discussion, the term ".IFcc" will be used to refer generically to any one of the directives above. The .IF directive requires exactly one operand. .IFNULL may have 0 or more operands. The remaining directives above may have one or two operands.

The .IFcc Directive declares the start of a block of lines to be conditionally assembled. Operands should evaluate to numeric values, except for .IFNULL, which can have any type of operand. Forward references are not permitted. Examples:

```
SERIAL.IO=1                     ;Assemble for system with serial I-O
        ...
        .IF     SERIAL.IO       ;Assemble only if using serial I-O
        ...
```

When the assembler encounters a .IF directive, it evaluates the operand expression. If the value of the operand is non-zero ("true"), assembly proceeds normally. If the operand evaluates to 0 ("false"), then the assembler "skips" all following statements until a .ENDIF or .ELSE directive is encountered. Skipped lines do not appear in the listing (but the .IF and .ENDIF do) and do not cause any object code to be generated. The assembler totally ignores skipped statements except that it increments its internal statement number counter for each line skipped, so that when assembly is resumed the line number printed on the listing will include the number of lines skipped. The .ENDIF and .ELSE directives which terminate the scope of the .IF are discussed below.

Conditional assembly is often used to allow one source program to be assembled for a variety of similar but non-identical systems. The portion of the program peculiar to each system is enclosed in a conditionally-assembled block where the .IFcc specifies the conditions under which the block should be assembled. An equate (or a direct definition, described later) at the beginning of the program can then be used to control which blocks should actually be assembled.

Conditional assembly directives may be nested up to seven deep. This means that a block of lines between and .IFcc and an .ENDIF can itself contain .IFcc directives.

The .IFLT, .IFLE, .IFNE, .IFEQ, .IFGE, and .IFGT directives may have either one or two operands. If only one operand is specified, the second operand defaults to zero. EXAMPLE:

```
        ...
        .IFLT   *,$2000
        ...
        .ELSE
        *=$4000
        ...
        .ENDIF
```

The conditional assembly above tests whether the code being assembled has reached address $2000 yet, and if it has, begins assembly at $4000.

```
        .IFGT   MAX-MIN
        ...
        .ENDIF
```

This example tests the value of (MAX-MIN) and assembles the following block if the result is greater than zero.

Probably the most valuable use of conditional assembly is in conjuction with macros. Additional examples are given in the section on macros, along with an explanation of the .IFNULL directive.

.ENDIF Directive

The .ENDIF directive is used to terminate a block of conditionally-assembled code. The .ENDIF directive does not have any operands and MUST NOT have a label. The .ENDIF must pair with a previously encountered .IFcc directive. In the case of nested .IFcc's, each .ENDIF encountered by the assembler will be associtated with the most recently encountered .IFcc. Example:

```
MTU130  =       0               ;SET TO 1 FOR MTU130 OR 0 FOR NON-MTU-130 ASSEMBLY
NOT     =       1
AIM     =       0               ;SET TO 1 FOR AIM SYSTEM
KIM     =       1               ;SET TO 1 FOR KIM SYSTEM
        ...
        .IF     NOT-MTU130
        ...
        .IF     AIM
        ...
        .ENDIF                  ;END OF CODE FOR AIM
        ...
        .IF     KIM
        ...
        .ENDIF                  ;END OF CODE FOR KIM
        ...
        .ENDIF                  ;END OF CODE FOR NON-MTU-130
        ...
```

.ELSE Directive

The .ELSE directive is used to declare the start of a block of conditionally assembled code which should be assembled under the opposite condition of that specified on the .IF above it. The .ELSE does not have any operand and MUST NOT have a label. Example:

```
REAL_MATH=0
        ...
        .IF     REAL_MATH       ;ASSEMBLE IF FLOATING POINT PRESENT
        ...
        .ELSE                   ;ASSEMBLE IF NO FLOATING POINT PRESENT
        ...
        .ENDIF
        ...
```

The .ELSE must be paired with a preceeding .IFcc and a following .ENDIF. Only one .ELSE may occur for each .IFcc. In the case of nested .IFcc directives, the .ELSE is always associated with the .IFcc which it most closely follows. The .ELSE is normally used to separate two mutually-exclusive blocks of conditionally assembled code. If the condition specified on the .IFcc is true, then the code between the .IFcc and the .ELSE will be assembled and the code between the .ELSE and the .ENDIF will not be assembled. If the condition specified on the .IFcc is false, then the code between the .IFcc and the .ELSE will not be assembled and the code between the .ELSE and the .ENDIF will be assembled.

MACROS

INTRODUCTION

Macros provide a kind of shorthand for the assembly language programmer. A macro is a user-defined mnemonic instruction which "stands for" one more other machine instructions or directives. A macro is invoked when its name appears in the opcode field of an instruction. When the assembler recognizes the name as a macro, it substitutes the desired lines of code in its place. Before a macro can be invoked, it must be defined. A macro definition begins with the .MACRO directive which specifies the name of the macro, and ends with an .ENDMAC directive. For example:

```
LSRA4   .MACRO
        LSR     A
        LSR     A
        LSR     A
        LSR     A
        .ENDMAC
```

defines a macro called "LSRA4", which shifts the accumulator 4 bits to the right. Once defined, the macro can be invoked, for example:

```
OUTH2   LDA     VALUE           ;GET BYTE TO OUTPUT
        LSRA4                   ;MOVE HI NYBBLE TO LOW NYBBLE
        JSR     OUTNYB          ;CONVERT NYBBLE TO ASCII HEX AND OUTPUT
        LDA     VALUE           ;RECALL BYTE
        AND     #$0F            ;GET LOW NYBBLE
        JSR     OUTNYB          ;OUTPUT LOW NYBBLE
        ...
```

When the assembler encounters this text, it will automatically replace the LSRA4 line with the four lines of the macro body. The resulting expansion will be:

```
OUTH2   LDA     VALUE           ;GET BYTE TO OUTPUT
        LSR     A
        LSR     A
        LSR     A
        LSR     A
        JSR     OUTNYB          ;CONVERT NYBBLE TO ASCII HEX AND OUTPUT
        LDA     VALUE           ;RECALL BYTE
        AND     #$0F            ;GET LOW NYBBLE
        JSR     OUTNYB          ;OUTPUT LOW NYBBLE
        ...
```

Once defined, macros may be invoked as many times as desired. The lines contained in the macro body can include directives as well as actual machine instructions, and can even invoke other macros. In addition, macros can be invoked with arguments specified in the operand field, which are substituted at selected places in the text when it is expanded. This is a very powerful tool for many applications. Conditional assembly is important within macros because it allows generation of different code depending on the nature of the arguments passed. These capabilities will be explained in detail in the following section. Appendix E contains a complete program using macros. Appendix F explanins how to use the supplied file of macro definitions to simulate an assembler for 8080 or Z-80 microprocessors.

.MACRO and .ENDMAC Directives

A macro definition begins with a .MACRO directive and ends with a .ENDMAC directive. The label field of the .MACRO directive must contain a legal identifier which is the desired name of the macro. The name may duplicate an ordinary label, but not a built-in opcode. If the same name is used for a normal label and a macro, the assembler knows which you mean by the way it is used, since a normal label can never appear in the opcode field. Therefore the following is legal:

```
        ...
THERE   .MACRO
        ...
        .ENDMAC
        ...
THERE   LDA     #0
        ...
```

However, the following will be flagged as a duplicate label error, because JSR is a standard machine opcode mnemonic:

```
        ...
JSR     .MACRO
        ...
        .ENDMAC
        ...
```

When the assembler encounters a .MACRO directive, it copies all following lines into a table in memory without further processing (except for discarding comments) until an .ENDMAC pseudo is encountered. This forms the macro body definition. If no matching .ENDMAC is found, the assembler simply keeps copying text until end of file or until it runs out of memory! If you ever get a a message indicating an assembly abort due to symbol table overflow of the heap, it is most likely because of a missing .ENDMAC. A macro must be defined before it is invoked. Macro definitions may not be nested; however, a macro invocation may occur inside a macro definition. A macro invocation within a macro definition may refer to a macro which is subsequently defined, since it is not really invoked until the outermost macro is invoked. Macros may also invoke themselves, as will be shown later.

Any number of macros may be defined, limited only by the size of the symbol table. Macros may not be redefined. An attempt to define the same macro twice will give a duplicate label error.

ARGUMENT PASSING

Argument passing is what makes macros really useful. When the macro is invoked, one or more arguments may be specified in the operand field. When the assembler expands the macro, it will automatically substitute the arguments in desired locations in the macro body. The location of the desired replacement must be shown in the macro definition by an exclamation point followed by the number of the desired argument. Thus !1 will be replaced by the first argument, !2 by the second argument, etc. The following example illustrates argument passing:

```
DISPLAY .MACRO
        SVC     2
        .BYTE   2,$0D,!1,0
        .ENDMAC
        ...
        DISPLAY 'MTU MACRO ASSEMBLER'
        DISPLAY '    VERSION 1.1'
        ...
```

When the assembler encounters the DISPLAY mnemonic in the opcode field, it substitutes the macro body text, replacing the "!1" with the entire string of text in the operand field (including the quote characters). The resulting expansions generate:

```
        SVC     2
        .BYTE   2,$0D,'MTU MACRO ASSEMBLER',0
        SVC     2
        .BYTE   2,$0D,'    VERSION 1.1',0
```

Multiple arguments should be separated by commas. Many programmers like to put a comment after the .MACRO mnemonic which lists the expected arguments. When a macro is invoked, the number of arguments specified may be less than, equal to, or greater than the number of arguments actually referred to in the body of the macro. If for example, a macro definition contains references to !1, !2, and !3, but only one argument is specified when the macro is invoked, then !2 and !3 will simply be replaced with nothing (called null). A null argument is not the same as a blank, which occupies one character instead of zero. Excess arguments are ignored. If more than 9 arguments are needed, !A through !Z may be used for the tenth through 36th arguments. Substitutions are always made, even inside a quoted string. If you need to have the character "!" appear literally inside a macro definition, use "!!" instead. The assembler will substitute a single "!" for it. Substitutions are not made in comments in a macro body because the assembler discards all comments when the macro is defined. When passing long arguments, care should be exercised that no lines will exceed 80 characters after all substitutions are made. The following macro illustrates some of these points:

```
ERRMSG  .MACRO                  ; ERRMSG   CHANNEL,OFFENDER
        SVC     2
        .BYTE   !1,$0D,'ERROR!!  ILLEGAL !2',0
        ...
        ERRMSG  6,WIDGET
        ...
```

This ivocation of ERRMSG expands to:

```
        SVC     2
        .BYTE   6,$0D,'ERROR!  ILLEGAL WIDGET',0
```

Occasionally you may wish to pass an argument with a comma in it. This can be done by either enclosing the argument in quotes or in curly brackets. The difference is that when an argument enclosed in quotes is passed, the quotes are included in the substitution, but curly brackets are not. For example, another invocation of the ERRMSG macro above might be:

```
        ERRMSG   2, {SCREW, 4-40, 1/2 INCH}
```

Enclosing the argument in curly brackets tells the assembler that there is only two arguments, not four.

CONDITIONAL ASSEMBLY IN MACROS

Conditional assembly is frequently used inside macros. Typically, different code is assembled depending on the passed arguments. Conditional assembly can also be used to test for the existence of arguments. The special symbol !# in a macro definition is replaced by the actual number of arguments passed when the macro is invoked. The .IFNULL directive can be used to test for a null argument. A null argument can be passed in the middle of a list by simply having two commas together. For example, the following macro outputs a message specified as the first argument over the channel specified as the second argument. If no second argument is specified, channel 2 is used by default:

```
MSG .MACRO                      : MSG 'TEXT'  ,CHANNEL
    SVC 2
    .IFNULL     !2
    .BYTE       2,!1,0
    .ELSE
    .BYTE       !2,!1,0
    .ENDIF
    .ENDMAC
```

LOCAL LABELS

Consider the following macro, which increments a two-byte variable specified as the argument:

```
INCW    .MACRO                  ;  INCW  WORDVARIABLE
        INC     !1
        BNE     DONE
        INC     !1+1
DONE
        .ENDMAC
```

Now suppose we invoke this macro twice:

```
        INCW    PTR1
        INCW    PTR2
```

The first invocation will work fine, but the second invocation will produce a duplicate label error, because the label DONE will appear again in the label field. One way to work around this problem is to pass the label (or part of the label) as an argument, and require that the programmer provide a new argument for each invocation, for example:

```
INCW    .MACRO                  ;   INCW  WORDVARIABLE,NEWLABEL
        INC     !1
        BNE     DONE.!2
        INC     !1+1
DONE.!2
        .ENDMAC
        ...
        INCW    PTR1,1
        INCW    PTR2,2
```

The resulting labels will be DONE.1 and DONE.2 for the respective invocations, avoiding the duplication problem. A better solution is to let the assembler generate the unique label.

The assembler keeps an internal counter of macro invocations. Each time a macro is invoked, it increments this 3 digit counter. If the special symbol !!! appears anywhere in the body of a macro, it is automatically replaced with the current invocation number. If more than 999 macros are invoked, the three exclamation points are replaced with a four digit number (starting at 1000). All local labels will appear in the symbol table printout at the end of the listing. Here is the improved macro:

```
INCW    .MACRO                  ;   INCW  WORDVARIABLE
        INC     !1
        BNE     DN!!!
        INC     !1+1
DN!!!
        .ENDMAC
```

Each time this macro is invoked, a new number is substituted for !!!, avoiding duplicate labels. If another macro is invoked between references to a local symbol, the assembler will keep track of the correct number. For example:

```
INCWIF  .MACRO                  ;INCWIF FLAG,WORDVARIABLE (INCREMENT IF FLAG NOT 0)
        LDA     !1
        BEQ     DN!!!
        INCW    !2
DN!!!
```

This macro invokes our previously defined macro INCW. Note that even though it uses the same label, DN!!!, and even though the invocation of INCW will increment the assembler's internal invocation number, the BEQ will still branch to the intended label. Also notice that when INCWIF is invoked, whatever argument is passed to it as the second argument is passed though to INCW as the its first argument. It is therefore quite feasable to build up more complex macros which invoke simpler macros. Here is an example:

```
TROUBLE .MACRO                  ; TROUBLE NAME,LIMITNAME
        SVC     2
        .BYTE   2,$0D,'***ERROR: !1 REACHED LIMIT (!2)',0
        .ENDMAC
;
INC_CHK .MACRO  ;INC_CHK   VARIABLE,LIMIT  ,ERROR_RECOVERY
;   THIS MACRO PERFORMS A DOUBLE PRECISION INCREMENT ON THE VARIABLE
;   GIVEN AS THE 1ST ARG., CHECKS IT AGAINST A LIMIT VARIABLE GIVEN
;   AS THE 2ND ARG., AND BRANCHES TO THE ERROR RECOVERY IF THE
;   LIMIT IS EXCEEDED.  IF NO ERROR-RECOVERY ADDRESS IS SPECIFIED,
;   AN INLINE MESSAGE IS GENERATED INSTEAD.
        INCW    !1              ;BUMP WORD POINTER
        LDA     !1
        CMP     !2
        LDA     !1+1            ;CHECK AGAINST LIMIT VALUE
        SBC     !2+1
        BCC     DN!!!           ;BRANCH IF OK
        .IFGT   !#,2            ;IF THIRD ARG SPECIFIED
        JMP     !3              ;JUMP TO ERROR RECOVERY
        .ELSE
        TROUBLE !1              ;ELSE ISSUE MESSAGE
        .ENDIF
DN!!!
        .ENDMAC
```

The invocation,

```
        INC_CHK  PTR1,PTR1LIM
```

will be expanded by the assembler to generate this code:

```
        INC     PTR1
        BNE     DN002
        INC     PTR1+1
DN002
        LDA     PTR1
        CMP     PTR1LIM
        LDA     PTR1+1
        SBC     PTR1LIM+1
        BCC     DN001
        SVC     2
        .BYTE   2,$0D,'***ERROR: PTR1 REACHED LIMIT (PTR1LIM)',0
DN001
```

RECURSIVE MACROS

Besides invoking other macros, a macro can also invoke itself! This is occasionally useful. For example, here is a recursive macro defining a multi-bit left shift:

```
ASLN    .MACRO                  ; ASLN  OPERAND,COUNT
        ASL     !1
        .IFGT   !2,1
        ASLN    !1,!2-1
        .ENDIF
        .ENDMAC
```

If this macro is invoked by:

```
        ASLN    A,4
```

it will generate code to perform four left shifts on the accumulator. Remember however when using recursive macros that macros may only be invoked eight deep.

PRINTING EXPANDED MACROS

Normally you will not want the expanded lines of the macro to appear in your listing, only the invocation. This helps make assembly listings more readable. However, you may force the assembler to print all expanded lines by using the .LIST directive with an operand of 5 (or 7). You will be able to identify the expansion lines in the listing easily because the line number in the leftmost column will not change. Using .LIST with an operand of 1 turns off macro expansion listing (this is the default). For example:

```
        .LIST   5               ;show all macro expansions
        ...
        .LIST   1               ;don't show macro expansions
        ...
```

See the description of .LIST for more details.

RUNNING THE ASSEMBLER

The assembler is invoked from CODOS with a command of the form:

MACASM ⟨source⟩ [⟨keyword⟩=⟨value⟩...] [⟨identifier⟩==⟨expression⟩...] .

Only the first argument is required, and must be the source file name. If no extension is given, ".A" will be assumed. The default drive is drive 0. For example:

```
MACASM MYPROG
```

will assemble the file MYPROG.A on drive 0. If no further arguments are given, the listing and object code will be output to files with the same name but ".C" and ".L" file extensions, respectively. Therefore, the object code output for this example will be a file called MYPROG.C and the listing will be output to a file called MYPROG.L.

Other arguments may follow the source file name. In this case, arguments are specified using the form, ⟨keyword⟩ = ⟨value⟩ , where keywords are chosen from the list below.

O=⟨name⟩ Object code file (standard CODOS loadable file). If ommitted, defaults to ⟨name⟩.C, where ⟨name⟩ is the source file name. O=N can be specified to inhibit object code generation. Object code can only be generated on a disk file. It cannot be placed directly in memory. If no file extension is given, .C will be used. The default drive is the same drive that the source file is on.

L=⟨name⟩ Listing file. If omitted, defaults to ⟨name⟩.L. The default extension is .L and the default drive is the same drive as the source file. L=N can be used to inhibit listing. L=P specifies the printer. In this case it is your responsibility to assure that the proper printer driver is present and that the P device is defined (see CODOS manual section 10). L=C specifies the console. If your system has another output device you may specify it.

X=⟨name⟩ External definitions file. If ommitted, defaults to no external definitions. The Externals file contains definitions which are to be available during the assembly; normally, it is the definitions file from prior assemblies or a list of global identifier equates, used to link programs together. Only one Externals file may be specified. The use of external and definitions files are explained later.

D=⟨name⟩ Definitions file. If ommitted, defaults to no definitions file. The Definitions file is written by the assembler and will contain equate overrides for all identifiers declared in .DEF directives in the program being assembled. Normally, it is used to facilitate linking of separately-assembled programs.

E=⟨name⟩ Error file. If omitted, defaults to the Console, and can be eliminated by specifying E=N. The error file will be written to contain each erroneous source line and the accompanying error message. Errors are also described on the listing file. If a file name is specified, the default extension is .E and the default drive is the same as the source file.

M=N Suppress generation of cross-reference map. If not specified, a cross reference map will be generated with the symbol table at the end of the listing. Suppressing the map reduces symbol table requirements by 4 bytes per symbol reference.

S=N Suppress printing of symbol table. If ommitted, a sorted symbol table will be generated at the end of the listing. If specified, no symbol table or cross-reference map will be generated. Specifying S=N reduces the time required for assembly and shortens the listing file.

When specifying file names for the optional arguments described above, the ditto mark (") may be used as a kind of "shorthand" to specify the same filename as the source file, up to but not including the ".".

EXAMPLES OF ASSEMBLER COMMAND LINES

```
MACASM GRAPHICS
```

uses GRAPHICS.A:0 as the source file and will generate an object file GRAPHICS.C:0 and a listing file GRAPHICS.L:0.

```
MACASM FOURIER:1 O=":0
```

assembles FOURIER.A:1 with a listing file FOURIER.L:1 and an object file FOURIER.C:0.

```
MACASM YOURS.A  L=N O=HIS:1
```

assembles YOURS.A:0 and produces an object file HIS.C on drive 1. No listing is produced.

```
MACASM MYBIGGIE:1 E=MYBIGERRORS L=N X=LIB1
```

assembles MYBIGGIE.A:1 with no listing, object file MYBIGGIE.C:1, an error summary on MYBIGERRORS.E:1, and will read the external definitions from LIB1.A:1.

```
MACASM MUSIC.T:1 L=":0 D="DEFS M=N
```

assembles MUSIC.T:1 with listing MUSIC.L:0 (without cross-reference map), object MUSIC.C:1, and will write the definitions file MUSICDEFS.A:1.

CONSOLE DISPLAY DURING ASSEMBLY

Once the MACASM command is executed, the assembler will "sign on" and display a message such as:

```
MTU 6502 MACRO ASSEMBLER VERSION 1.1 (21-OCT-82)
Copyright 1982 Micro Technology Unlimited
SOURCE FILE IS MYPROG.A
```

This indicates the name of the file the assembler is currently reading. If you have specified an external file (X) or have any .READ directives in your file, the assembler will issue a similar message each time it changes the source file. If any errors are detected, the offending line will be displayed on the screen with a brief error message below it (unless the E option has been used to redirect or cancel the error summary).

The assembler actually reads the source file twice. Each reading is called a pass. After pass 1 the assembler will display "0 ERRORS IN PASS 1" and go on to pass 2. If any errors were detected during pass 1, pass 2 will not be run. However, if the listing is enabled, pass 1 will be run again to produce the listing. Assuming no errors were found in pass 1, the assembler will display the names of any file changes made during pass 2. At the end of pass 2, there will be a pause for a few seconds with no disk activity, while the assembler sorts the symbol table in memory. Then a final error summary and signoff message will be displayed. The signoff message gives a summary of the number of source lines, the total number of identifiers encountered, and the total number of references to these identifiers. These numbers are given in decimal. It also displays the amount of symbol table storage used for the assembly and the total amoutn which was available in memory. This number is given in hexadecimal. The symbol table is the area of memory where the assembler stores the definitions for all the symbols used in the program. For a small program, the amount of symbol table space used may seem overly large. This is because the assembler also stores the definitions for all the machine language mnemonic opcodes and assembler directives in the symbol table. If your system is equipped with a DATAMOVER or other memory expansion memory board in bank 3, MACASM will display the amount of memory used in this bank also. The assembler automatically tests for the existence of RAM in bank 3, and if present, MACASM will store all cross-referecnes and macro definitions there. This allows for assembly of very large programs. The assembler has a large enough symbol table to handle about 1900 typical symbols with the cross reference map enabled, if expansion RAM is present, or about 1100 symbols without expansion RAM.

DIRECT DEFINITIONS AND EQUATE OVERRIDES

Besides specifying the files and options above, the CODOS command line can also be used to pass one or more direct definitions to the assembler. These direct definitions appear similar to the keyword arguments, but use two equals signs, "==". The keywords used are the names of identifiers to be defined, and are interpreted by the assembler as "equate overrides". Example:

```
MACASM STUFF.A  L=0 PROG.ORG==$2000 DATA.ORG==$A0
```

assembles the source file STUFF.A without a listing, using an object file STUFF.C. In addition, the assembler will proceed as though the two lines,

```
PROG.ORG   ==    $2000
DATA.ORG   ==    $A0
```

existed at the top of the source file. The "==" operator is called an "equate override" and is used to override any definition elsewhere in the program using an ordinary equate (=). This is most often used to relocate a program without having to edit the source file. Continuing with the example above, suppose that the program being assembled contained the following fragment:

```
          ...
PROG.ORG=$B400
DATA.ORG=0
          ...
          *=DATA.ORG    ;ZERO PAGE ORIGIN
          ...
          *=PROG.ORG    ;PROGRAM ORIGIN
          ...
```

Normally, this program would be assembled with a segment starting at $0000 and a segment starting at $B400. However, the direct definitions specified on the command line override the equates for PROG.ORG and DATA.ORG, so that the the program will instead be assembled with segments at $00A0 and $2000. With proprer planning of your equates, this capability makes it very easy to relocate your program or make other minor changes without having to modify the source program in any way.

Equate overrides may also be entered into the text of a source program. Normally, an equate override is used to supercede a normal equate, but there is no requirement that an ordinary equate also exist. Equate overrides may not be used to override the definitions of anything other than an equate (e.g., you cannot override a label), and the equate override must precede the ordinary equate it overrides. A normal equate which is expected to be overridden is often called the "default definition", because it is the equate which will be used if no other is specified as an override. A common use of direct definitons is to specify the value to be used on the operand of a .IF directive. For example:

```
    MACASM INVENTORY SYSTEM==PET
```

can be used to define which conditional assembly blocks are to be assembled in the following program fragment:

```
    ...
;
MTU130=1              ;IDENTIFIERS FOR TARGE MACHINE
AIM=2
PET=3
;
SYSTEM=MTU130         ;DEFAULT DEFINITION FOR CONDITIONAL ASSY.
    ...
    .IFEQ     SYSTEM,MTU130
    ...
    .ENDIF
    .IFEQ     SYSTEM,PET
    ...
    .ENDIF
    ...
```

LINKING TO SEPARATELY ASSEMBLED PROGRAMS

Often one program needs to refer to identifiers in another program. For example, a program may contain subroutine calls (JSRs) to subroutines in a library of assembly-language subroutines (not to be confused with libraries used by BASIC). One way to handle this is to include the library source file in the assembly by using a .READ directive. However, this may present problems if the library routines contain identifiers which you have inadvertantly used in the main program for a different purpose. Also, including many .READs may make the assembly process unduly long and possibly even overflow the symbol table. What is really needed in this case is the ability to only refer to selected symbols in the library routines. In order to accomplish this, the MTU assembler provides a simple and effective way for programs to "link" to separately-assembled programs.

DEFINITIONS FILE AND EXTERNALS FILE

There are two steps required to link to a separately-assembled library program. First, you must tell the assembler what identifiers in the library are to be available to other programs. This is accomplished by using .DEF directives in the file to declare all "global" identifiers, and by specifying the name of the file which is to hold these definitions. The D= name option should be specified on the assembly command line; otherwise, the definitions specified will not be saved on a file. When the assembler completes the assembly of the program, it will write the definitions of all the specified identifiers on the definitions file as a series of equate overrides.

A separately assembled program may now use these definitions by merely specifying the name of the file using the X option on the MACASM command line. The file name specified by the X keyword is called the externals file. Alternatively, the file can be read using .READ directives from within the program. The external file of one program is the definitions file from the previously assembled program. This program can in turn specify its own file of definitions for use by yet another separate assembly. It can also specify the same file as it is using for an externals file, in which case the new definitions will be appended on the end of the file. These techniques make interfacing a large assembly-language project quite feasable. The following example program segments illustrate the use of the definitions file and the externals file in linking to a separate assembly.

Assume that the following program fragment represents a file of three utility subroutines which are to be called from separately-assembled programs:

```
            ...
            .DEF    DRAW,MOVE,ERASE
            ...
DRAW        LDA     #1
            ...
            RTS
;
MOVE        LDA     #0
            ...
            RTS
;
ERASE       LDA     #$FF
            ...
            RTS
            .END
```

This library is assembled with the following command:

```
MACASM GRAPHSUB  D=GRAPHSUBDEFS
```

Now a separately assmbled program can refer to these routines by name by specifying the externals file previously defined:

```
MACASM DRAWBOX X=GRAPHSUBDEFS
```

This program may now contain references to DRAW, MOVE and ERASE without internally defining their value, for example:

```
          ...
DOBOX     JSR     MOVE
          LDX     #45
          JSR     DRAW
          ...
```

Once this program is assembled, you may load and run the library and the main program with the following CODOS commands:

```
GET GRAPHSUB
DRAWBOX
```

Alternatively, you could GET both files and use the CODOS SAVE command to store both programs together as a single file.

OBJECT CODE OUTPUT

The assembler outputs object code during pass 2. Object code is generated even if errors are detected during pass 2. Therefore you will generally want to delete the object file and listing before reassembling after an error.

The object code is generated as a standard CODOS loadable program file. You may execute the program by merely typing its name once assembly is done, or the program may be just loaded into memory using the GET command. The GETLOC command can be used to determine what memory locations are occupied by the generated code. See the CODOS manual for details.

If paper-tape format (hexadecimal) object code is desired, the GET command should be used to load the program into memory and the SERDMP utility used to generate the paper-tape format ouput. See the Utilities manual for details of operaion of SERDMP.

PROGRAM LISTING

A sample program listing output from the MTU assembler follows the source file in Appendix E. On each line of the listing, the assembler outputs a source line from the program read, with additional information in the first 18 columns. This information consists of the statement number (in decimal), the address where any code generated from that line will begin, and the object code generated. The addresses and object code are shown in hexadecimal. If any errors are discovered by the assembler, it prints an error message under the offending line, for example:

```
 0025 B420     BEGIN   JST    DOIT
ERROR 0001 ***************^ UNDEFINED
```

In this case, the assembler objected to the undefined opcode "JST", (which presumably should have been a "JSR"). The position of the up-arrow at the end of the row of asterisks does not necessarily indicate the exact location of the problem. It only indicates how much of the line the assembler has actually "looked at" before it issued the error message . The error may be anywhere before that point, or may even be caused by an error earlier in the program. Error numbers are simply assigned sequentially by the assembler as detected. For instance, if you are examining error number 3 you know that there are two errors listed prior to this point in the listing. All lines flagged on the listing will also be displayed on the console (unless the E option is used to redirect or cancel the error summary). Usually you will not even have to consult the listing to fix errors turned up by the assembler.

The assembler actually reads the entire source program twice. Normally, no listing is produced until pass 2. If errors are found by the assembler in pass 1 of the assembly, then no second pass is made. Instead, pass one is "rerun" to generate the listing (if not suppressed). In this case, no object code will be shown since object code is generated only during pass 2.

If you have a line with a character string operand of more than two characters, the object code column of the listing will only show the first two bytes of the object code, with a ".." following to indicate more code not listed. Equates have an "=" after the address field to indicate that the value shown is the value of the equate and not the program counter address. Similarly, equate overrides show "==" after the address field. If a "----" appears in the address field of an equate, it indicates that the assembler ignored the equate due to an earlier equate override defining the same identifier.

The assembler normally inserts a form feed character into the listing every 57 lines to cause a printer to eject to the top of the next page, or whenever a .PAGE is encountered. The page number is printed on the last three characters of the header line. MACASM also prints the most current title (specified on the .PAGE directive) in the first 50 columns, and the date in columns 66-75.

At the end of the listing the assembler prints the symbol table and cross reference map. If the map is disabled, the assembler will print four symbols per line instead of the normal one symbol per line. Each symbol is followed by its hexadecimal definition. If the map is enabled, this is then followed by a list of all line numbers which refer to the symbol. You will find this map very useful for locating all the places that a given variable is used, or all the places that a certain subroutine is called.

THE DISASM UTILITY

The DISASM Utility program is a 6502 disassembler which runs in the Utility program space($B400-B7FF). It will disassemble machine code starting at a given address, and has two modes of specifying how much to disassemble. The first is the line mode, where the amount to disassemble is specified in lines of output. The second is the starting and ending address mode, where the address range to disassemble is given. In either mode, output to a channel of printer may be specified.

SYNTAX - LINE MODE:

DISASM ⟨starting address⟩ [⟨L ⟨lines⟩ [⟨out⟩]]

ARGUMENTS:

⟨starting address⟩ = the hex address at which to start disassembling.
⟨lines⟩ = number of lines, i.e. instructions, to disassemble. Defaults to 22 lines.
⟨out⟩ = channel or "P" for the printer device to send the disassembly output. Defaults to the console. The "L" parameter must be specified before an output channel or device may be specified.

DISCUSSION:

If no 'L' parameter is given, the number of lines defaults to 22. After disassembling the specified number of lines, the disassembler will wait for a key to be press on the keyboard. If this key is a CTL-Q, another group of lines will be disassembled. If it is something else, the disassembler will prompt with "DISASM " and wait for another set of disassembly parameters. A carriage return at this point will return to CODOS.

If a device is specified, it will be assigned to channel 6. At no point will the disassembler free this or any other channel. You must free this channel yourself when you return to CODOS.

EXAMPLES:

```
DISASM 1000
```

will disassemble 22 lines of the machine code at hex address 1000.

```
DISASM 2000 L60 P
```

will disassemble 60 lines of the machine code at 2000 to the printer.

SYNTAX - START/END MODE:

DISASM ⟨starting address⟩ ⟨ending address⟩ [⟨out⟩]

ARGUMENTS:

⟨starting address⟩ = the hex address at which to start disassembling.
⟨ending address⟩ = the hex address at which to stop disassembling.
⟨out⟩ = channel or "P" for the printer device to send the disassembly output. Defaults to the console.

DISCUSSION:

In this mode, the range of addresses specified will be disassembled, including the machine code instruction which spans the ending address. When done, the disassembler will prompt with "DISASM)" and wait for another set of disassembly parameters. A carriage return at this point will return to CODOS.

If a device is specified, it will be assigned to channel 6. At no point will the disassembler free this or any other channel. You must free this channel yourself when you return to CODOS.

EXAMPLES:

DISASM 4060 4090

will disassemble the machine language instructions from 4060 hex through 4090 hex. The disassembly will be output to the console.

DISASM 8000 8400 4

will disassemble the machine language instructions from 8000 hex through 8400 hex. The disassembly will be output to channel 4.

NOTES:

1. While disassembling to the console, CTL-S and CTL-Q may be used to temporarily stop and resume the disassembly output, respectively. CTL-C will abort back to CODOS at any point.

2. "P" is the only output device accepted. Specifying other devices will result in an ILLEGAL CHANNEL NUMBER error.

APPENDIX A

ASSEMBLER ERROR MESSAGES

Error Messages Which Abort Assembly Immediately

SYMBOL TABLE FULL (HASH). The assembler cannot proceed because there are more identifiers in the program than the assembler can handle. Suggestion: split the program into two or more smaller programs and assemble separately using Definitions and Externals files for program linkage.

SYMBOL TABLE FULL (HEAP). The assembler cannot proceed because the area of memory allocated for storing identifiers' names and values is full. Often caused by a missing .ENDMAC. If program is very large, try suppressing the cross reference map using M=N and rerun (or better, add expansion RAM in bank 3).

COMMAND SYNTAX. The ASM command line contains an error. Suggestion: compare your command line with the sample command lines given in this manual, correct, and rerun.

NO SOURCE. The file specified as a source file was not found by the assembler. Suggestion: Check the spelling and drive number of the file name. Remember that the assembler assumes a default extension of .A.

OUTPUT FILE EXISTS. The listing or object file (or both) already exist on the specified or default drive. Suggestion: DELETE the file and rerun.

MACROS OVERNESTED. A macro has been invoked more than 8 deep or has passed too many arguments to fit in the assembler's argument table.

Error Messages Which Do Not Abort Assembly Immediately

SYNTAX. The line contains a syntax error. Suggestion: check for illegal punctuation, illegal addressing modes, or missing quote marks. Consult the Instruction set summary in Appendix B or the Wirth syntax diagrams in Appendix C.

ARITHMETIC. The result of an arithmetic operation has overflowed or is undefined. Suggestion: check for division or remainder by 0.

IDENTIFIER. The identifier is illegal. Suggestion: Check for non-uppercase letters. Also remember that anything starting in column 1 must be a label or a comment.

0-PAGE. A zero page operand is required but was not found. Suggestion: check the value of the identifier and move it into zero page if necessary. If the value is a zero page value, then this is a forward reference to a zero page value, which is not permitted. Move the definition of the operand to an earlier point in the program.

DUPLICATE. The same identifier has already been previously defined. Suggestion: use the cross-reference map to locate the earlier definition and correct.

UNDEFINED. An identifier is referenced but is not defined in the program. Suggestion: check for spelling errors or an ommitted definition.

COMPLEXITY. The operand contains too many levels of brackets. Suggestion: use an equate to define part of the expression.

ADDRESSING. An illegal addressing mode is indicated for the opcode specified. Suggestion: Consult the list of legal addressing modes in Appendix B.

CONDITIONAL. A conditional assembly error has been detected. Suggestion: check for missing .IF or .ENDIF, label on .ELSE or .ENDIF, more than one .ELSE, or nesting greater than 7 deep.

RANGE. An addressing range error has occurred. Suggestion: if this is a conditional branch, then the destination for the branch is more than +125 or less than -128 bytes from the present instruction. If this instruction is an indirect jump, then the address crosses a page boundary, which will result in an error due to a hardware bug in all 6502 processors.

FILE. The specified file name is illegal. Suggestion: check for missing colon between the file name and drive number, illegal file extension, incorrectly spelled file name or file name greater than 12 characters.

MACRO. A problem has been encountered with a macro definition. Check for invocations exceeding 8 deep.

APPENDIX B

6502 INSTRUCTION SET SUMMARY

Instruction	Opcode	Cycles	Bytes	Description	Addressing Mode
ADC #n	69	2	2	Add A with carry	Immediate
ADC n	65	3	2	Add A with carry	Zero page
ADC n,X	75	4	2	Add A with carry	Zero page,X
ADC (n,X)	61	6	2	Add A with carry	Indirect,X
ADC (n),Y	71	5+	2	Add A with carry	Indirect,Y
ADC nn	6D	4	3	Add A with carry	Absolute
ADC nn,X	7D	4+	3	Add A with carry	Absolute,X
ADC nn,Y	79	4+	3	Add A with carry	Absolute,Y
AND #n	29	2	2	AND A	Immediate
AND n	25	3	2	AND A	Zero page
AND n,X	35	4	2	AND A	Zero page,X
AND (n,X)	21	6	2	AND A	Indirect,X
AND (n),Y	31	5+	2	AND A	Indirect,Y
AND nn	2D	4	3	AND A	Absolute
AND nn,X	3D	4+	3	AND A	Absolute,X
AND nn,Y	39	4+	3	AND A	Absolute,Y
ASL A	0A	2	1	Arithmetic shift left	Accumulator
ASL n	06	5	2	Arithmetic shift left	Zero page
ASL n,X	16	6	2	Arithmetic shift left	Zero page,X
ASL nn	0E	6	3	Arithmetic shift left	Absolute
ASL nn,X	1E	7	3	Arithmetic shift left	Absolute,X
BCC r	90	2+	2	Branch if carry is clear	Relative
BCS r	B0	2+	2	Branch if carry is set	Relative
BEQ r	F0	2+	2	Branch if equal	Relative
BNE r	D0	2+	2	Branch if not equal	Relative
BMI r	30	2+	2	Branch if minus	Relative
BPL r	10	2+	2	Branch if plus	Relative
BVC r	50	2+	2	Branch if overflow clear	Relative
BVS r	70	2+	2	Branch if overflow set	Relative
BIT nn	2C	4	3	Bit test	Absolute
BIT n	24	3	2	Bit test	Zero page
BRK	00	7	1	Break	Implied
CLC	18	2	1	Clear carry	Implied
CLD	D8	2	1	Clear decimal	Implied
CLI	58	2	1	Clear interrupt disable	Implied
CLV	B8	2	1	Clear overflow	Implied
CMP #n	C9	2	2	Compare A	Immediate
CMP n	C5	3	2	Compare A	Zero page
CMP n,X	D5	4	2	Compare A	Zero page,X
CMP (n,X)	C1	6	2	Compare A	Indirect,X
CMP (n),Y	D1	5+	2	Compare A	Indirect,Y
CMP nn	CD	4	3	Compare A	Absolute
CMP nn,X	DD	4+	3	Compare A	Absolute,X
CMP nn,Y	D9	4+	3	Compare A	Absolute,Y

Instruction	Opcode	Cycles	Bytes	Description	Addressing Mode
CPX #n	E0	2	2	Compare X	Immediate
CPX n	E4	3	2	Compare X	Zero page
CPX nn	EC	4	3	Compare X	Absolute
CPY #n	C0	2	2	Compare Y	Immediate
CPY n	C4	3	2	Compare Y	Zero page
CPY nn	CC	4	3	Compare Y	Absolute
DEC n	C6	5	2	Decrement	Zero page
DEC n,X	D6	6	2	Decrement	Zero Page,X
DEC nn	CE	6	3	Decrement	Absolute
DEC nn,X	DE	7	3	Decrement	Absolute,X
DEX	CA	2	1	Decrement X	Implied
DEY	88	2	1	Decrement Y	Implied
EOR #n	49	2	2	Exclusive OR A	Immediate
EOR n	45	3	2	Exclusive OR A	Zero page
EOR n,X	55	4	2	Exclusive OR A	Zero page,X
EOR (n,X)	41	6	2	Exclusive OR A	Indirect, X
EOR (n),Y	51	5+	2	Exclusive OR A	Indirect Y
EOR nn	4D	4	3	Exclusive OR A	Absolute
EOR nn,X	5D	4+	3	Exclusive OR A	Absolute,X
EOR nn,Y	59	4+	3	Exclusive OR A	Absolute,Y
INC n	E6	5	2	Increment	Zero page
INC n,X	F6	6	2	Increment	Zero Page,X
INC nn	EE	6	3	Increment	Absolute
INC nn,X	FE	7	3	Increment	Absolute,X
INX	E8	2	1	Increment X	Implied
INY	C8	2	1	Increment Y	Implied
JMP nn	4C	3	3	Jump	Absolute
JMP (nn)	6C	5	3	Jump	Indirect
JSR nn	20	6	3	Jump to subroutine	Absolute
LDA #n	A9	2	2	Load A	Immediate
LDA n	A5	3	2	Load A	Zero page
LDA n,X	B5	4	2	Load A	Zero page,X
LDA (n,X)	A1	6	2	Load A	Indirect, X
LDA (n),Y	B1	5+	2	Load A	Indirect Y
LDA nn	AD	4	3	Load A	Absolute
LDA nn,X	BD	4+	3	Load A	Absolute,X
LDA nn,Y	B9	4+	3	Load A	Absolute,Y
LDX #n	A2	2	2	Load X	Immediate
LDX n	A6	3	2	Load X	Zero Page
LDX n,Y	B6	4	2	Load X	Zero page,Y
LDX nn	AE	4	3	Load X	Absolute
LDX nn,Y	BE	4+	3	Load X	Absolute Y

Instruction	Opcode	Cycles	Bytes	Description	Addressing Mode
LDY #n	A0	2	2	Load Y	Immediate
LDY n	A4	3	2	Load Y	Zero page
LDY n,X	B4	4	2	Load Y	Zero page,X
LDY nn	AC	4	3	Load Y	Absolute
LDY nn,X	BC	4+	3	Load Y	Absolute,X
LSR A	4A	2	1	Logical shift right	Accumulator
LSR n	46	5	2	Logical shift right	Zero page
LSR n,X	56	6	2	Logical shift right	Zero page,X
LSR nn	4E	6	3	Logical shift right	Absolute
LSR nn,X	5E	7	3	Logical shift,right	Absolute,X
NOP	EA	2	1	No operation	Implied
ORA #n	09	2	2	OR A	Immediate
ORA n	05	3	2	OR A	Zero page
ORA n,X	15	4	2	OR A	Zero page,X
ORA (n,X)	01	6	2	OR A	Indirect, X
ORA (n),Y	11	5+	2	OR A	Indirect Y
ORA nn	0D	4	3	OR A	Absolute
ORA nn,X	1D	4+	3	OR A	Absolute,X
ORA nn,Y	19	4+	3	OR A	Absolute,Y
PHA	48	3	1	Push A	Implied
PHP	08	3	1	Push Status	Implied
PLA	68	4	1	Pull A	Implied
PLP	28	4	1	Pull Status	Implied
ROL A	2A	2	1	Rotate left through carry	Accumulator
ROL n	26	5	2	Rotate left through carry	Zero page
ROL n,X	36	6	2	Rotate left through carry	Zero page,X
ROL nn	2E	6	3	Rotate left through carry	Absolute
ROL nn,X	3E	7	3	Rotate left through carry	Absolute,X
ROR A	6A	2	1	Rotate rt. through carry	Accumulator
ROR n	66	5	2	Rotate rt. through carry	Zero page
ROR n,X	76	6	2	Rotate rt. through carry	Zero page,X
ROR nn	6E	6	3	Rotate rt. through carry	Absolute
ROR nn,X	7E	7	3	Rotate rt. through carry	Absolute,X
RTI	40	6	1	Return from interrupt	Implied
RTS	60	6	1	Return from subroutine	Implied
SBC #n	E9	2	2	Subtract A with Borrow	Immediate
SBC n	E5	3	2	Subtract A with Borrow	Zero page
SBC n,X	F5	4	2	Subtract A with Borrow	Zero page,X
SBC (n,X)	E1	6	2	Subtract A with Borrow	Indirect, X
SBC (n),Y	F1	5+	2	Subtract A with Borrow	Indirect Y
SBC nn	ED	4	3	Subtract A with Borrow	Absolute
SBC nn,X	FD	4+	3	Subtract A with Borrow	Absolute,X
SBC nn,Y	F9	4+	3	Subtract A with Borrow	Absolute,Y

Instruction	Opcode	Cycles	Bytes	Description	Addressing Mode
SEC	38	2	1	Set carry	Implied
SED	F8	2	1	Set decimal mode	Implied
SEI	78	2	1	Set interrupt disable	Implied
STA n	85	3	2	Store A	Zero page
STA n,X	95	4	2	Store A	Zero page,X
STA (n,X)	81	6	2	Store A	Indirect,X
STA (n),Y	91	6	2	Store A	Indirect,Y
STA nn	8D	4	3	Store A	Absolute
STA nn,X	9D	5	3	Store A	Absolute,X
STA nn,Y	99	5	3	Store A	Absolute,Y
STX n	86	3	2	Store X	Zero page
STX n,Y	96	4	2	Store X	Zero page,Y
STX nn	8E	4	3	Store X	Absolute
STY n	84	3	2	Store Y	Zero page
STY n,X	94	4	2	Store Y	Zero page,X
STY nn	8C	4	3	Store Y	Absolute
TAX	AA	2	1	Transfer A to X	Implied
TAY	A8	2	1	Transfer A to Y	Implied
TSX	BA	2	1	Transfer S to X	Implied
TXA	8A	2	1	Transfer X to A	Implied
TXS	9A	2	1	Transfer X to S	Implied
TYA	98	2	1	Transfer Y to A	Implied

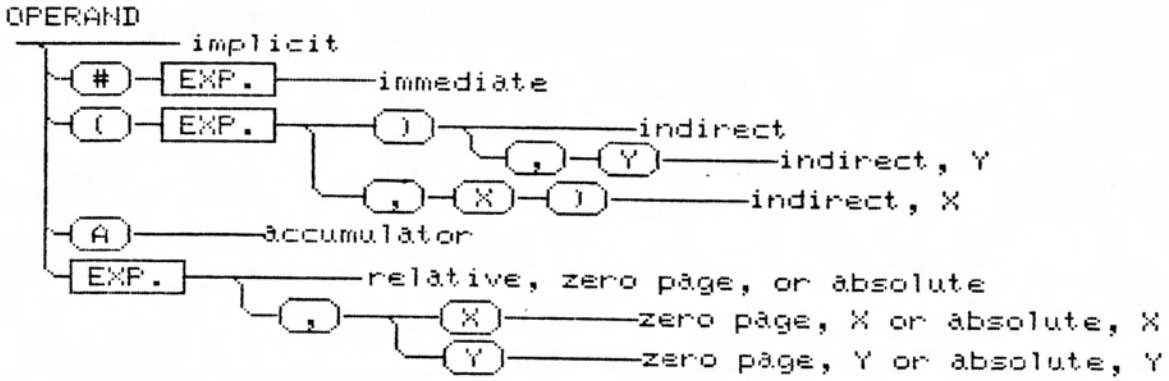
Notation used in the Table

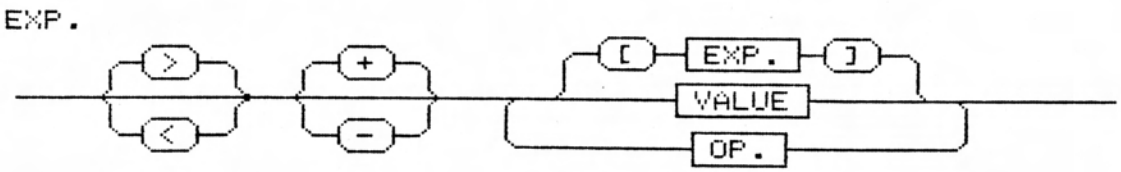
n indicates a zero page expression.
nn indicates a non-zero page expression.
r indicates an absolute expression which evaluates to within +125 or -128 bytes of the current program counter.

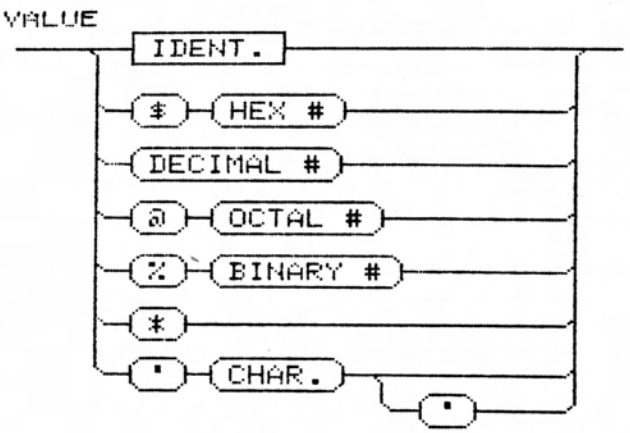
\+ in the Cycles column indicates possible additional cycles as follows:
For branches:
Add 1 if branch taken.
Add 2 if branch taken accross page boundary.
For other instructions:
Add 1 if indexing accross page boundary.

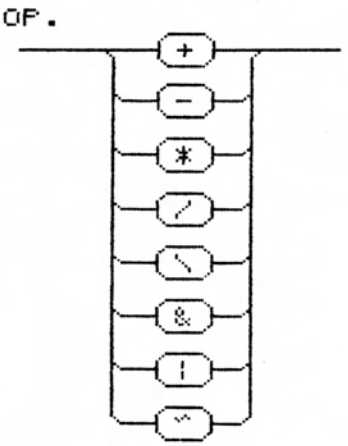
APPENDIX C

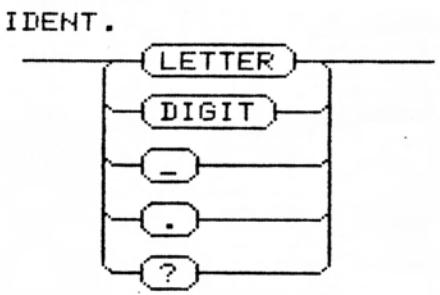
WIRTH SYNTAX DIAGRAMS FOR ASSEMBLER OPERANDS











Note: The syntax diagrams above were produced using the MTU-130 GREDIT Demo program and printed on an Epson MX-70 printer.

APPENDIX D

MEMORY USAGE AND ASSEMBLER MODIFICATION

The MTU-130 assembler uses the following memory locations:

0040-0073	Scratch zero page RAM.
0700-33FF	Variables and Executable code.
3400-34FF	Direct definitions buffer.
3500-35FF	Object code buffer.
3600-39FF	Source program buffer.
3A00-3EFF	Listing buffer.
3F00-ABFF	Symbol table primary heap (names, definitions, etc.)
AC00-BBFF	Symbol table hash table (pointers into heap).
BC00-BDFF	Macro invocation argument stack.
0000-FEFF:3	Secondary heap (macro defs., cross-refs., if RAM is present).

If no expansion RAM is present in bank 3, the macro definitions and cross-references are put in the primary heap. MACASM detects automatically if RAM is present in bank 3. If desired you may limit the amount of bank 3 RAM which can be used by MACASM by changing the upper limit pointer at $0718 in bank 0. Changing $0718 to $0000 will leave bank 3 RAM untouched by MACASM.

In addition, the assembler requires the presence of CODOS, the SVC processor and the standard I-O drivers for proper operation.

Advanced users may wish to modify certain assembler buffer locations or parameters. No assurances of proper operation can be given if these values are changed. The following addresses are provided for this purpose:

0706	Pointer to start of source buffer. Source buffer is relocatable.
0708	Pointer to end+1 of source buffer. Must start/end on page bound.
070A	Pointer to start of listing buffer. Listing buffer is relocatable.
070C	Pointer to end+1 of listing buffer. Must start/end on page bound.
070E	Pointer to start of heap for symbol table. Note that the first $040A bytes of the heap are initialized to contain the built-in opcodes and directives. This code is relocatable on page bound.
0710	Pointer to end+1 of heap for symbol table. Must end on page bound.
0712	Pointer to start of hash table. Hash table is relocatable.
0714	Pointer to end+1 of hash table. Alter hash table size only if also changing location $0720 below.
0716	Pointer to start of secondary heap in bank 3.
0718	Pointer to end+1 of secondary heap in bank 3 ("sniff" limit)
071A	Pointer to start of macro argument stack. Relocatable.
071E	Pointer to end+1 of macro argument stack.
0720	Hash table size code: $0F=4K, $07=2K, $03=1K.
0724	Value to be substituted for undefined operand (default=FFFF). Do not alter to smaller than $0100.
0727	Number of lines per page on listing. Set to 0 for continuous. Default is 57 decimal ($39).
0728	Maximum column to start name on in symbol table listing (default=61).
0729	Distance between tabs in symbol table listing (default=20).
072A	Maximum column to start a cross reference number in on list (def.=75).

072B ASCII character to be used for exclusive OR operator.
072C ASCII character to be used for OR operator.
072D ASCII character to be used for AND operator.
072E ASCII character to be used for remainder operator.
072F ASCII character to be used for division operator.
0730 ASCII character to be used for multiplication operator (not PC sym.)
0731 ASCII character to be used for subtraction operator.
0732 ASCII character to be used for addition operator.
0733 ASCII character to be used for left bracket operator.
0734 ASCII character to be used for right bracket operator.
0735 ASCII character to be used for low byte operator.
0736 ASCII character to be used for high byte operator.
0737 ASCII character to be used for hex prefix.
0738 ASCII character to be used for octal prefix.
0739 ASCII character to be used for binary prefix.
073A ASCII character to be used for quote character.
073B ASCII control character for tab.
073C ASCII character to be used as macro argument lead-in (normally "!").
073D ASCII character to be used as macro arg. left string delimiter.
073E ASCII character to be used as macro arg. right string delimiter.
073F ASCII character to be used after "!" in macro definitions which is to be replaced by the number of arguments actually present ("#").
0740-0742 Three ASCII characters permitted to be embedded in identifiers.
0743 ASCII character to be used for program counter symbol.
0744 ASCII character to be used for comment character.
0745 Flag. Change to $80 to disable the first form feed (only) on the listing (saves paper if you always remember to initialize your printer to top-of-form before printing the listing).
0634-0638 Table of command keyword letters for list, object, errors, externals, and definitions (default=L,O,E,X,D).
07F0-07F2 Default drive numbers for source, listing, and object file, respectively (all default to 0).
07FC-0801 Default file extensions for source, listing, object, errors, externals, and definitions files, respectively.

097C ASCII form-feed character ($0C). Change to CR ($0D) to cause two carriage returns in lieu of a formfeed on every new listing page. (See address $745 above to disable first formfeed only).

All addresses are subject to change in future revisions.

APPENDIX E

Sample Assembly Program Source File

```
          .PAGE   'CONCAT UTILITY: JOIN FILES TOGETHER'
LIST      =       1       ;SPECIFY LIST==7 TO FORCE FULL LIST
          .LIST   LIST
;
; PROGRAM TO CONCATENATE FILES BY LARRY ISAACS

; DECLARATIONS...

BUF       = $A000         ;LOCATION OF LARGE TRANSIENT BUFFER
BUFEND    = $B3FF
BUFSIZ    = BUFEND-BUF
U1        = $B2           ;CODOS STUFF
U2          = $B4
U5          = $BA
SVCENB      = $EE
;
CODORG    = $B400

;          MACROS...

MSG       .MACRO  ;'TEXT OF MESSAGE'
          SVC     2
          .BYTE   2,!1,$0D,0
          .ENDMAC
;
ERROR      .MACRO  ;'TEXT OF ERROR MESSAGE'
           SVC     2
           .BYTE   2,!1,$0D,0
           SEC
           RTS
           .ENDMAC

SETIM      .MACRO  ; WORDCONSTANT,WORDDEST
           LDA     #<!1
           STA     !2
           LDA     #>!1
           STA     !2+1
           .ENDMAC

          *= CODORG
;
CONCAT    SEC             ;SET SVC ENABLE
          ROR SVCENB
          LSR FINFLG      ;CLEAR FINISHED FLAG
          SVC 12          ;QUERRY I/O BUFFERS
          LDX #6
          STX CHAN
          JSR GETFN1      ;TRY TO GET CHANNEL FROM COMMAND LINE
          BCC CONCA2      ;BR IF GOT ONE
          LSR FINFLG      ;CLEAR AGAIN
CONCA1    MSG 'FILE TO CONCATENATE TO?'
          LDX #6
          JSR GETFN       ;GET THE FILE NAME
          BCC CONCA2      ;BR IF GOT ONE
```

```
        BIT FINFLG      ;SEE IF TIME TO QUIT
        BPL CONCA1      ;BR IF NO, TRY AGAIN
        SVC 1
CONCA2  BPL CONCA3      ;BR IF NEW FILE
        SVC 18          ;POSITION TO EOF IF NOT
CONCA3  ASL A           ;CHECK FOR LOCKED FILE
        ASL A
        BPL CONCA5      ;BR IF NOT LOCKED
        MSG 'FILE IS LOCKED.'
        JMP CONCA1
CONCA4  JSR XFER        ;TRANSFER THE DATA
CONCA5  MSG 'FILE TO CONCATENATE?'
        LDX #5
        JSR GETFN       ;GET INPUT FILE
        BCC CONCA4      ;BR IF GOT ONE
        BIT FINFLG      ;SEE IF TIME TO QUIT
        BPL CONCA5      ;BR IF NO, TRY AGAIN
        LDX #5          ;CLOSE INPUT FILE IF ASSIGNED
        SVC 14
        BCC CONCA6      ;BR IF NOT ASSIGNED
        SVC 22
CONCA6  LDX #6          ;CLOSE OUTPUT IF ASSIGNED
        SVC 14
        BCC CONCA7
        SVC 22
CONCA7  RTS             ;ALL DONE
;
; GET A FILE NAME.
; IF ASSIGNING TO CHANNEL 6, MAKE SURE NOT LOCKED
; IF ASSIGNING TO CHANNEL 5, MAKE SURE IT EXISTS
;
GETFN   STX CHAN        ;SAVE THE CHANNEL TO ASSIGN TO
        LDX #1
        SVC 5           ;INPUT A FILE NAME
        BCC GETFN1      ;BR IF GO SOMETHING
GETFND  ROR FINFLG      ;SET FINISHED FLAG
        JMP NOFILE
GETFN1  LDA (U5),Y      ;SKIP SPACES
        CMP #$20
        BNE GETFN2
        INY
        BNE GETFN1      ;BR ALWAYS
GETFN2  CMP #$0D        ;SEE IF NO FILE NAME
        BEQ GETFND
        SVC 29          ;CHECK THE FILE NAME
        BCS GETFNE      ;BR IF DEVICE NAME
        STA OUSTAT      ;SAVE STATUS
        BIT OUSTAT      ;SET FLAGS
        BVS GETDOE      ;BR IF DRIVE NOT OPEN
        LDX CHAN        ;CHECK CHANNEL
        CPX #6
        BNE GETFN3      ;BR IF NOT OUTPUT CHANNEL
        AND #$0C        ;SEE IF WRITE PROTECTED DISK
        CMP #$40
        BEQ GETWPE      ;BR IF YES
GETFN3 CPX #5
```

```
        BNE GETFN4
        LDA #$20        ;MAKE SURE INPUT FILE EXISTS
        BIT OUSTAT
        BEQ GETFXE      ;BR IF DOESN'T EXIST
GETFN4  LDA OUSTAT      ;ASSIGN THE FILE
        AND #$03
        SVC 21
        CLC
        RTS
NOFILE  SEC
        RTS
GETFNE  ERROR 'FILE NAME ERROR.'
GETDOE  ERROR 'DRIVE NOT OPEN.'
GETWPE  ERROR 'DISK IS WRITE PROTECTED'
GETFXE  ERROR 'FILE NOT FOUND.'
;
; TRANSFER ALL OF INPUT FILE TO OUTPUT FILE
;
XFER    SETIM BUF,U1
        SETIM BUFSIZ,U2
        LDX #5          ;READ FROM FILE
        SVC 15
        BCS XFERF       ;BR IF EOF
        SETIM BUF,U1
        LDX #6          ;WRITE TO FILE
        SVC 16
        JMP XFER        ;BACK FOR MORE
XFERF   RTS
;
; VARIABLES SPACE
;
FINFLG  *=*+1
CHAN    *=*+1
OUSTAT  *=*+1
        .END
```

```
CONCAT UTILITY: JOIN FILES TOGETHER          MTU 6502 MACASM 1.1 21-OCT-82   1

0001 0000                    .PAGE   'CONCAT UTILITY: JOIN FILES TOGETHER'
0002 0001  =      LIST     =       1       ;SPECIFY LIST==7 TO FORCE FULL LIST
0003 0000                    .LIST   LIST
0004 0000         ;
0005 0000         ; PROGRAM TO CONCATENATE FILES BY LARRY ISAACS
0006 0000
0007 0000         ; DECLARATIONS...
0008 0000
0009 A000  =      BUF      = $A000         ;LOCATION OF LARGE TRANSIENT BUFFER
0010 B3FF  =      BUFEND   = $B3FF
0011 13FF  =      BUFSIZ   = BUFEND-BUF
0012 00B2  =      U1       = $B2           ;CODOS STUFF
0013 00B4  =      U2         = $B4
0014 00BA  =      U5         = $BA
0015 00EE  =      SVCENB     = $EE
0016 0000         ;
0017 B400  =      CODORG   = $B400
0018 0000
0019 0000         ;         MACROS...
0020 0000
0021 0000         MSG      .MACRO  ;'TEXT OF MESSAGE'
0022 0000                  SVC     2
0023 0000                  .BYTE   2,!1,$0D,0
0024 0000                  .ENDMAC
0025 0000         ;
0026 0000         ERROR     .MACRO  ;'TEXT OF ERROR MESSAGE'
0027 0000                   SVC     2
0028 0000                   .BYTE   2,!1,$0D,0
0029 0000                   SEC
0030 0000                   RTS
0031 0000                   .ENDMAC
0032 0000
0033 0000         SETIM     .MACRO  ; WORDCONSTANT,WORDDEST
0034 0000                   LDA     #(!1
0035 0000                   STA     !2
0036 0000                   LDA     #)!1
0037 0000                   STA     !2+1
0038 0000                   .ENDMAC
0039 0000
0040 0000                  *= CODORG
0041 B400         ;
0042 B400 38      CONCAT   SEC              ;SET SVC ENABLE
0043 B401 66EE             ROR SVCENB
0044 B403 4E69B5           LSR FINFLG       ;CLEAR FINISHED FLAG
0045 B406 000C             SVC 12           ;QUERRY I/O BUFFERS
0046 B408 A206             LDX #6
0047 B40A 8E6AB5           STX CHAN
0048 B40D 20A6B4           JSR GETFN1       ;TRY TO GET CHANNEL FROM COMMAND LINE
0049 B410 902D             BCC CONCA2       ;BR IF GOT ONE
0050 B412 4E69B5           LSR FINFLG       ;CLEAR AGAIN
0051 B415         CONCA1   MSG 'FILE TO CONCATENATE TO?'
0052 B431 A206             LDX #6
0053 B433 2097B4           JSR GETFN        ;GET THE FILE NAME
0054 B436 9007             BCC CONCA2       ;BR IF GOT ONE
0055 B438 2C69B5           BIT FINFLG       ;SEE IF TIME TO QUIT
```

```
CONCAT UTILITY: JOIN FILES TOGETHER          MTU 6502 MACASM 1.1 21-OCT-82   2

0056 B43B 10D8           BPL CONCA1      ;BR IF NO, TRY AGAIN
0057 B43D 0001           SVC 1
0058 B43F 1002   CONCA2  BPL CONCA3      ;BR IF NEW FILE
0059 B441 0012           SVC 18          ;POSITION TO EOF IF NOT
0060 B443 0A     CONCA3  ASL A           ;CHECK FOR LOCKED FILE
0061 B444 0A             ASL A
0062 B445 101A           BPL CONCA5      ;BR IF NOT LOCKED
0063 B447                MSG 'FILE IS LOCKED.'
0064 B45B 4C15B4         JMP CONCA1
0065 B45E 2043B5 CONCA4  JSR XFER        ;TRANSFER THE DATA
0066 B461        CONCA5  MSG 'FILE TO CONCATENATE?'
0067 B47A A205           LDX #5
0068 B47C 2097B4         JSR GETFN       ;GET INPUT FILE
0069 B47F 90DD           BCC CONCA4      ;BR IF GOT ONE
0070 B481 2C69B5         BIT FINFLG      ;SEE IF TIME TO QUIT
0071 B484 10DB           BPL CONCA5      ;BR IF NO, TRY AGAIN
0072 B486 A205           LDX #5          ;CLOSE INPUT FILE IF ASSIGNED
0073 B488 000E           SVC 14
0074 B48A 9002           BCC CONCA6      ;BR IF NOT ASSIGNED
0075 B48C 0016           SVC 22
0076 B48E A206   CONCA6  LDX #6          ;CLOSE OUTPUT IF ASSIGNED
0077 B490 000E           SVC 14
0078 B492 9002           BCC CONCA7
0079 B494 0016           SVC 22
0080 B496 60     CONCA7  RTS              ;ALL DONE
0081 B497        ;
0082 B497        ; GET A FILE NAME.
0083 B497        ; IF ASSIGNING TO CHANNEL 6, MAKE SURE NOT LOCKED
0084 B497        ; IF ASSIGNING TO CHANNEL 5, MAKE SURE IT EXISTS
0085 B497        ;
0086 B497 8E6AB5 GETFN   STX CHAN        ;SAVE THE CHANNEL TO ASSIGN TO
0087 B49A A201           LDX #1
0088 B49C 0005           SVC 5           ;INPUT A FILE NAME
0089 B49E 9006           BCC GETFN1      ;BR IF GO SOMETHING
0090 B4A0 6E69B5 GETFND  ROR FINFLG      ;SET FINISHED FLAG
0091 B4A3 4CE0B4         JMP NOFILE
0092 B4A6 B1BA   GETFN1  LDA (U5),Y      ;SKIP SPACES
0093 B4A8 C920           CMP #$20
0094 B4AA D003           BNE GETFN2
0095 B4AC C8             INY
0096 B4AD D0F7           BNE GETFN1      ;BR ALWAYS
0097 B4AF C90D   GETFN2  CMP #$0D        ;SEE IF NO FILE NAME
0098 B4B1 F0ED           BEQ GETFND
0099 B4B3 001D           SVC 29          ;CHECK THE FILE NAME
0100 B4B5 B02B           BCS GETFNE      ;BR IF DEVICE NAME
0101 B4B7 8D6BB5         STA OUSTAT      ;SAVE STATUS
0102 B4BA 2C6BB5         BIT OUSTAT      ;SET FLAGS
0103 B4BD 703A           BVS GETDOE      ;BR IF DRIVE NOT OPEN
0104 B4BF AE6AB5         LDX CHAN        ;CHECK CHANNEL
0105 B4C2 E006           CPX #6
0106 B4C4 D006           BNE GETFN3      ;BR IF NOT OUTPUT CHANNEL
0107 B4C6 290C           AND #$0C        ;SEE IF WRITE PROTECTED DISK
0108 B4C8 C940           CMP #$40
0109 B4CA F043           BEQ GETWPE      ;BR IF YES
0110 B4CC E005   GETFN3  CPX #5
```

```
CONCAT UTILITY: JOIN FILES TOGETHER          MTU 6502 MACASM 1.1 21-OCT-82   3

0111 B4CE D007           BNE GETFN4
0112 B4D0 A920           LDA #$20        ;MAKE SURE INPUT FILE EXISTS
0113 B4D2 2C6BB5         BIT OUSTAT
0114 B4D5 F056           BEQ GETFXE      ;BR IF DOESN'T EXIST
0115 B4D7 AD6BB5 GETFN4  LDA OUSTAT      ;ASSIGN THE FILE
0116 B4DA 2903           AND #$03
0117 B4DC 0015           SVC 21
0118 B4DE 18             CLC
0119 B4DF 60             RTS
0120 B4E0 38     NOFILE  SEC
0121 B4E1 60             RTS
0122 B4E2        GETFNE  ERROR 'FILE NAME ERROR.'
0123 B4F9        GETDOE  ERROR 'DRIVE NOT OPEN.'
0124 B50F        GETWPE  ERROR 'DISK IS WRITE PROTECTED'
0125 B52D        GETFXE  ERROR 'FILE NOT FOUND.'
0126 B543        ;
0127 B543        ; TRANSFER ALL OF INPUT FILE TO OUTPUT FILE
0128 B543        ;
0129 B543        XFER    SETIM BUF,U1
0130 B54B                SETIM BUFSIZ,U2
0131 B553 A205           LDX #5          ;READ FROM FILE
0132 B555 000F           SVC 15
0133 B557 B00F           BCS XFERF       ;BR IF EOF
0134 B559                SETIM BUF,U1
0135 B561 A206           LDX #6          ;WRITE TO FILE
0136 B563 0010           SVC 16
0137 B565 4C43B5         JMP XFER        ;BACK FOR MORE
0138 B568 60     XFERF   RTS
0139 B569        ;
0140 B569        ; VARIABLES SPACE
0141 B569        ;
0142 B569        FINFLG  *=*+1
0143 B56A        CHAN    *=*+1
0144 B56B        OUSTAT  *=*+1
0145 B56C                .END
   0 ERRORS IN PASS 2
```

```
SYMBOL TABLE WITH CROSS REFERENCES              MTU 6502 MACASM 1.1 21-OCT-82   4

BUF     $A000  0009 0011 0129 0129 0134 0134
BUFEND  $B3FF  0010 0011
BUFSIZ  $13FF  0011 0130 0130
CHAN    $B56A  0047 0086 0104 0143
CODORG  $B400  0017 0040
CONCA1  $B415  0051 0056 0064
CONCA2  $B43F  0049 0054 0058
CONCA3  $B443  0058 0060
CONCA4  $B45E  0065 0069
CONCA5  $B461  0062 0066 0071
CONCA6  $B48E  0074 0076
CONCA7  $B496  0078 0080
CONCAT  $B400  0042
ERROR   MACRO  0026 0122 0123 0124 0125
FINFLG  $B569  0044 0050 0055 0070 0090 0142
GETDOE  $B4F9  0103 0123
GETFN   $B497  0053 0068 0086
GETFN1  $B4A6  0048 0089 0092 0096
GETFN2  $B4AF  0094 0097
GETFN3  $B4CC  0106 0110
GETFN4  $B4D7  0111 0115
GETFND  $B4A0  0090 0098
GETFNE  $B4E2  0100 0122
GETFXE  $B52D  0114 0125
GETWPE  $B50F  0109 0124
LIST    $0001  0002 0003
MSG     MACRO  0021 0051 0063 0066
NOFILE  $B4E0  0091 0120
OUSTAT  $B56B  0101 0102 0113 0115 0144
SETIM   MACRO  0033 0129 0130 0134
SVCENB  $00EE  0015 0043
U1      $00B2  0012 0129 0129 0134 0134
U2      $00B4  0013 0130 0130
U5      $00BA  0014 0092
XFER    $B543  0065 0129 0137
XFERF   $B568  0133 0138
   0 ERRORS IN PASS 2

SUMMARY:
145 LINES, 36 SYMBOLS, 103 REFERENCES.
SYMBOL TABLE USED $05F1 OF $6D00 BYTES AVAILABLE (BANK 0)
AND $01DF OF $FF00 IN EXPANSION RAM BANK.
* END OF ASSEMBLY. *
```

APPENDIX F

USING THE MACROS_8080 LIBRARY TO ASSEMBLE 8080 PROGRAMS

The file MACROS_8080.A on the MACASM distribution disk contains a source library of 78 macro definitions and 10 equates which will enable MACASM to assemble programs written for the Intel 8080 microprocessor. Since object code for the 8080 is compatible with the Zilog Z-80, it can also be used to assemble programs to run on Z-80 processors, such as on the MTU PROGRAMMOVER board for the MTU-130.

To assemble code for the 8080 or Z-80, a source program should be prepared in the usual manner using 8080 mnemonics for all machine opcodes and registers, BUT WITH A "." AFTER EVERY OPCODE AND REGISTER NAME. The macros have been defined with a period terminating each name so there will be no conflict with 6502 machine opcodes with identical names (such as JMP). The registers (A., B., C., D., H., L., M., SP., and PSW.) follow a similar convention to avoid conflict with the reserved 6502 symbol A (specifying the accumulator addressing mode). Use the normal MACASM directives for specifying the origin, reserving storage, etc.(or add your own macros defining the equivalent Intel mnemonics).

The sample listing on the following page illustrates the use of the macro file. The macro definitions themselves are not listed and the macros are not expanded in order to save space. Specifying .LIST 7 will show all generated code.

If you plan to actually run the assembled code on the PROGRAMMOVER's Z-80, don't forget to use a .BANK pseudo to specify the memory bank which is occupied by the PROGRAMMOVER's onboard 64K of memory.

Naturally using macros to simulate an 8080 assembler will not detect all the errors that a "real" 8080 assembler should. For example,

```
CALL. H.
```

will generate code without an error, even though the CALL instruction cannot legally specify a register as the operand. However it will detect the most common errors such as undefined operands and duplicate labels.

Sample 8080 Cross Assembly Listing

```
DEMO PROGRAM FOR 8080 MACRO LIBRARY          MTU 6502 MACASM 1.1 21-OCT-82   1

0001 0000                .PAGE   'DEMO PROGRAM FOR 8080 MACRO LIBRARY'
0002 0000        ;
0003 0000        ;       SAMPLE 8080 PROGRAM ASSEMBLED USING MACASM 6502
0004 0000        ;       ASSEMBLER WITH MACROS_8080.A LIBRARY.
0005 0000        ;
0006 0000        ;       PROGRAM IS 16 BY 16 BIT MULTIPLY ROUTINE
0007 0000        ;       EXTRACTED FROM "AN INTEGER MATH PACKAGE FOR THE 8080",
0008 0000        ;       BY BRUCE D. CARBREY, BYTE MAGAZINE, MAY 1981, WITH
0009 0000        ;       THE PERMISSION OF THE AUTHOR.
0010 0000        ;
0011 0000        ;       SUB. EMULT COMPUTES (HL)=(HL)*(DE)
0012 0000        ;       USING SIGNED INTEGER MATH, AND BRANCHES TO OVERFLOW
0013 0000        ;       IF THE RESULT EXCEEDS 16 BITS.  OPTIMIZED FOR SPEED.
0014 0000        ;       ON RETURN, ZERO AND SIGN FLAGS REFLECT RESULT VALUE.
0015 0000        ;       A REG CLOBBERED, ALL OTHERS RESTORED.
0016 0000        ;
0017 0000                .READ   MACROS_8080.A
0018 0000                .LIST   0
0374 0000                .LIST 1
0375 0000                *=      $1000
0376 1000        EMULT   PUSH.   B.
0377 1001                PUSH.   D.          SAVE REGS
0378 1002                CALL.   RSLTSIGN    FIND SIGN OF RESULT
0379 1005                XRA.    A.
0380 1006                ADD.    H.
0381 1007                JZ.     HLSMALL     BRANCH IF (HL) LESS THAN 8 BITS
0382 100A                XRA.    A.
0383 100B                ADD.    D.          ELSE OTHER OP MUST BE .LT. 8 BITS
0384 100C                CNZ.    OVERFLOW    ...OR AN OVERFLOW WOULD RESULT
0385 100F                XCHG.
0386 1010        HLSMALL MOV.    A.,L.       MOVE 8 BIT OPERAND TO A
0387 1011                LXI.    H.,0        INITIALIZE PARTIAL PRODUCT
0388 1014        XMLOOP  STC.
0389 1015                CMC.
0390 1016                RAR.                ROTATE MULTIPLIER RIGHT OFF END
0391 1017                JNC.    SHIFTOP     BRANCH IF 0 WAS SHIFTED OUT
0392 101A                DAD.    D.          ELSE ADD M'CAND TO P. PROD.
0393 101B                CC.     OVERFLOW
0394 101E        SHIFTOP XCHG.
0395 101F                DAD.    H.          SHIFT MULTIPLICAND LEFT 1 BIT...
0396 1020                CC.     OVERFLOW    ...CHECKING FOR OVERFLOW
0397 1023                XCHG.
0398 1024                ORA.    A.
0399 1025                JNZ.    XMLOOP      REPEAT TILL MULT IS 0
0400 1028                POP.    D.          RECALL REGS
0401 1029        SIGNRCL MOV.    A.,H.
0402 102A                RLC.                FINAL OVERFLOW CHECK
0403 102B                CC.     OVERFLOW
0404 102E                MOV.    A.,B.       RECALL SIGN BYTE
0405 102F                RAL.
0406 1030                CC.     COMP2       CHANGE SIGN IF RESULT IS -
0407 1033                POP.    B.          RECALL REG
0408 1034                JMP.    ESIGN       SET FLAGS TO REFLECT RESULT, RET.
0409 1037        ;
0410 1037        ;       SUB. RSLTSIGN - COMPUTE RESULT SIGN, ABS. VAL OF (HL)
```

```
0411 1037          ;
0412 1037          RSLTSIGN MOV.   B.,H.      SIGN OF 1ST OPERAND
0413 1038                   MOV.   A.,H.      TO B AND TO A
0414 1039                   RAL.
0415 103A                   CC.    COMP2      ABSOLUTE VALUE IF NEGATIVE
0416 103D                   XCHG.
0417 103E                   MOV.   A.,H.      SIGN BYTE TO A
0418 103F                   XRA.   B.         RESULT SIGN
0419 1040                   MOV.   B.,A.      TO B FOR TEMP SAVE
0420 1041                   MOV.   A.,H.      SIGN OF 2ND OP TO A
0421 1042                   RAL.
0422 1043                   JC.    COMP2      ABSOLUTE VALUE, RETURN
0423 1046                   RET.
0424 1047          ;
0425 1047          ;        SUB. ESIGN - SET FLAGS TO REFLECT RESULT
0426 1047          ;
0427 1047          ESIGN    XRA.   A.         CLEAR FLAGS
0428 1048                   ADD.   H.         SET FLAGS TO REFLECT HI BYTE
0429 1049                   RNZ.              RETURN IF HI BYTE NON-0
0430 104A                   ADD.   L.         ELSE SEE IF LOW IS 0 TOO
0431 104B                   RZ.               IF SO, RETURN
0432 104C                   XRA.   A.         ELSE FORCE FLAGS TO SHOW +
0433 104D                   INR.   A.
0434 104E                   RET.
0435 104F          ;
0436 104F          ;        SUB. COMP2: 2'S COMPLEMENT (HL) (CHANGE SIGN)
0437 104F          ;
0438 104F          COMP2    MOV.   A.,H.
0439 1050                   CMA.
0440 1051                   MOV.   H.,A.
0441 1052                   MOV.   A.,L.
0442 1053                   CMA.
0443 1054                   MOV.   L.,A.
0444 1055                   INX.   H.
0445 1056                   RET.
0446 1057          ;
0447 1057          ;        REPLACE THIS CODE WITH ERROR PROCESSING...
0448 1057          ;
0449 1057          OVERFLOW HLT.
   0 ERRORS IN PASS 2
```

8080 OBJECT CODE GENERATED

```
      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

1000  C5 D5 CD 37 10 AF 84 CA  10 10 AF 82 C4 57 10 EB
1010  7D 21 00 00 37 3F 1F D2  1E 10 19 DC 57 10 EB 29
1020  DC 57 10 EB B7 C2 14 10  D1 7C 07 DC 57 10 78 17
1030  DC 4F 10 C1 C3 47 10 44  7C 17 DC 4F 10 EB 7C A8
1040  47 7C 17 DA 4F 10 C9 AF  84 C0 85 C8 AF 3C C9 7C
1050  2F 67 7D 2F 6F 23 C9 76
```

REFERENCES

(1) Programming the 6502 (Third Ed.), by Rodney Zaks.

(2) 6502 Assembly Language Programming, by Lance A. Leventhal.

(3) 6502 Software Gourmet Guide and Cookbook, by Robert Findley.

(4) MCS 6500 Microcomputer Family Programming Manual, Publication #6500-50A, MOS Technology, Inc. (Commodore).

(5) 6502 Assembly Lanugage Subroutines, by Lance A. Leventhal and Winthrop Saville.